

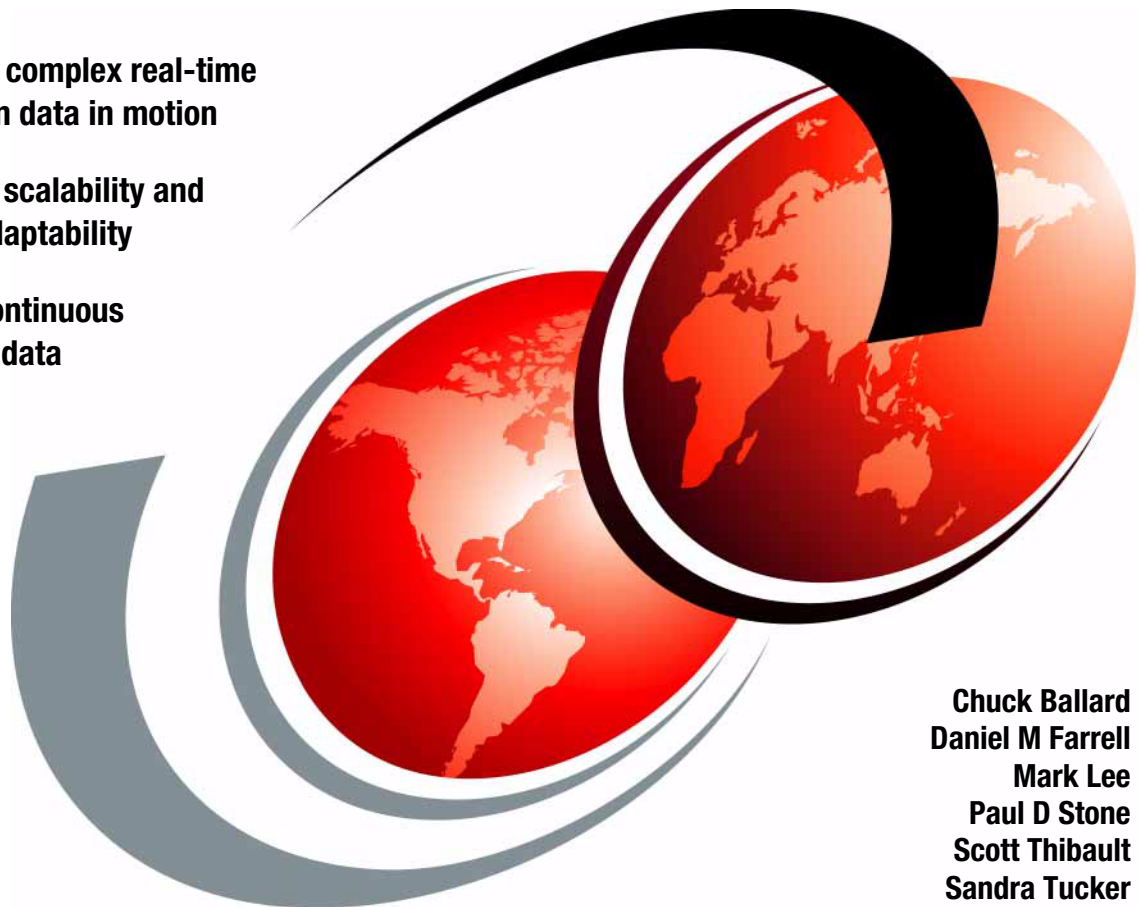
IBM InfoSphere Streams

Harnessing Data in Motion

Performing complex real-time analytics on data in motion

Supporting scalability and dynamic adaptability

Enabling continuous analysis of data



Chuck Ballard
Daniel M Farrell
Mark Lee
Paul D Stone
Scott Thibault
Sandra Tucker



International Technical Support Organization

IBM InfoSphere Streams: Harnessing Data in Motion

September 2010

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (September 2010)

This edition applies to Version 1.2.0 of IBM InfoSphere Streams (product number 5724-Y95).

© Copyright International Business Machines Corporation 2010. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Trademarks	x
 Preface	xi
The team who wrote this book	xii
Now you can become a published author, too!	xiv
Comments welcome	xv
Stay connected to IBM Redbooks	xv
 Chapter 1. Introduction	1
1.1 Stream computing	2
1.1.1 Business landscape	6
1.1.2 Information environment	9
1.1.3 The evolution of analytics	13
1.2 IBM InfoSphere Streams	16
1.2.1 Overview	18
1.2.2 Why IBM InfoSphere Streams	23
1.2.3 Example Streams implementations	26
 Chapter 2. IBM InfoSphere Streams concepts and terms	33
2.1 IBM InfoSphere Streams: Solving new problems	34
2.2 Concepts and terms	39
2.2.1 Streams Instances, Hosts, Host Types, and (Admin) Services	40
2.2.2 Projects, Applications, Streams, and Operators	44
2.2.3 Applications, Jobs, Processing Elements, and Containers	48
2.3 End-to-end example: Streams and the lost child	49
2.3.1 One application or many	51
2.3.2 Example Streams Processing Language code review	56
2.4 IBM InfoSphere Streams tools	64
2.4.1 Creating an example application inside Studio	65
2.4.2 Using streamtool	72
 Chapter 3. IBM InfoSphere Streams Applications	75
3.1 Streams Application design	76
3.1.1 Design aspects	77
3.1.2 Application purpose and classes of applications	78
3.1.3 Data sources	80
3.1.4 Output	83
3.1.5 Existing analytics	86

3.1.6	Performance requirements	87
3.2	Streams design patterns	88
3.2.1	Filter pattern: Data reduction	89
3.2.2	Outlier pattern: Data classification and outlier detection	92
3.2.3	Parallel pattern: High volume data transformation	96
3.2.4	Pipeline pattern: High volume data transformation	101
3.2.5	Alerting pattern: Real-time decision making and alerting	104
3.2.6	Enrichment pattern: Supplementing data	106
3.2.7	Unstructured Data pattern: Unstructured data analysis	109
3.2.8	Consolidation pattern: Combining multiple streams	112
3.2.9	Merge pattern: Merging multiple streams	115
3.2.10	Integration pattern: Using existing analytics	118
3.3	Using existing analytics	120
Chapter 4. Deploying IBM InfoSphere Streams Applications		127
4.1	Runtime architecture	128
4.2	Introduction to topologies	129
4.2.1	Single Host Instance	130
4.2.2	Multiple Hosts Instance	132
4.2.3	Starting and stopping the instance	134
4.3	Sizing the environment	136
4.4	Deployment implications for application design	137
4.4.1	Dynamic application composition	138
4.4.2	Overcoming performance bottleneck Operators	140
4.5	Application deployment strategy	144
4.5.1	Deploying Operators to specific nodes	145
4.6	Performance engineering	146
4.6.1	Define performance targets	148
4.6.2	Establish access to a suitable environment	149
4.6.3	Measure the performance and compare it to the target	150
4.6.4	Identify the dominant bottleneck	155
4.6.5	Remove the bottleneck	155
4.7	Security	156
4.7.1	Secure access	160
4.7.2	Security policies: Authentication and authorization	163
4.7.3	Processing Element containment	171
4.8	Failover, availability, and recovery	175
4.8.1	Recovering Operator state: Checkpointing	175
4.8.2	Recovering Processing Elements	176
4.8.3	Recovering application hosts	177
4.8.4	Recovering management hosts	178
4.9	Environment profiling	180

Chapter 5. IBM InfoSphere Streams Processing Language	187
5.1 Language elements	188
5.1.1 Structure of a Streams Processing Language program file	190
5.1.2 Streams data types	193
5.1.3 Stream schemas	194
5.1.4 Streams punctuation markers	196
5.1.5 Streams windows	196
5.1.6 Stream bundles	203
5.2 Streams Processing Language Operators	204
5.2.1 Operator usage language syntax	207
5.2.2 The Source Operator	208
5.2.3 The Sink Operator	211
5.2.4 The Functor Operator	212
5.2.5 The Aggregate Operator	214
5.2.6 The Split Operator	216
5.2.7 The Punctor Operator	220
5.2.8 The Delay Operator	220
5.2.9 The Barrier Operator	221
5.2.10 The Join Operator	222
5.2.11 The Sort Operator	224
5.3 The Preprocessor	226
5.4 Export and Import	233
5.4.1 Point to point	234
5.4.2 Publish and subscribe	236
5.5 Example Streams program	238
5.6 Debugging a Streams Application	239
5.6.1 Using Streams Studio to debug	241
5.6.2 Using streamtool to debug	245
5.6.3 Other debugging interfaces and tools	248
5.6.4 The Streams Debugger	249
Chapter 6. Advanced IBM InfoSphere Streams programming	257
6.1 Edge adaptors	258
6.1.1 DB2 and Informix Dynamic Server database Operators	258
6.1.2 The solidDBEnrich Operator	261
6.1.3 The WFOSource Operator	262
6.1.4 The InetSource Operator	263
6.2 User-defined functions	264
6.2.1 Using templates	266
6.2.2 Using libraries	267
6.3 User-defined Source Operators	267
6.3.1 Example user-defined Source Operator	268
6.3.2 Multiple streams	271

6.3.3 Unstructured data	272
6.4 User-defined Sink Operators	272
6.4.1 Example user-defined Sink Operator	273
6.4.2 Unstructured data	276
6.5 User-defined Operators.	276
6.5.1 An example user-defined Operator	276
6.5.2 Multiple streams	280
6.5.3 Unstructured data	281
6.5.4 Reusing user-defined Operators	282
6.5.5 Type-generic Operators	283
6.5.6 Port-generic Operators	285
6.5.7 Multi-threaded Operators	286
6.6 User-defined Built-in Operators.	289
6.6.1 Overview	289
6.6.2 Create the UDOP directory structure	290
6.6.3 Generate starting templates	290
6.6.4 Create a test application	291
6.6.5 Modify the starting templates	291
6.6.6 An example UBOP	295
6.6.7 Syntax specification	299
6.7 Hardware acceleration	299
6.7.1 Integrating hardware accelerators.	300
6.7.2 Line-speed processing	302
Appendix A. IBM InfoSphere Streams installation and configuration . .	303
Physical installation components of Streams	304
Installing Streams on single-tier development systems	305
Appendix B. Toolkits and samples	313
Overview	314
What is a toolkit or sample asset	314
Why provide toolkit or sample assets	314
Where to find the toolkits and sample assets	315
Currently available toolkit assets.	316
Streams Mining Toolkit.	317
Streams and data mining	317
Streams Mining Toolkit V1.2	318
How does it work.	318
Financial Markets Toolkit	319
Why Streams for financial markets	319
Financial Markets Toolkit V1.2	320
What is included in the Financial Markets Toolkit	321
Glossary	323

Abbreviations and acronyms 325

Related publications 327

Other publications 327

Online resources 327

IBM Education Support 327

 IBM Training 328

 Information Management Software Services 328

 IBM Software Accelerated Value Program 329

How to get Redbooks 330

Help from IBM 331

Index 333

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:


This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

DB2®
IBM®
Informix®
InfoSphere™
Passport Advantage®

RAA®
Redbooks®
Redpaper™
Redbooks (logo) ®
Smart Traffic™

Smarter Planet™
solidDB®
Solid®
WebSphere®

The following terms are trademarks of other companies:

Java, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Preface

In this IBM® Redbooks® publication, we discuss and describe the positioning, functions, capabilities, and programming best practices for IBM InfoSphere™ Streams (Streams).

Stream computing is a relatively new paradigm. In traditional data processing, you typically run queries against relatively static sources of data, which then provide you with a query result set for your analysis. With stream computing, you execute a process that can be thought of as a continuous query, that is, the results are continuously updated as the data sources are refreshed or extended over time. So, traditional queries seek and access static data to be queried and analyzed. But with stream computing, continuous streams of data flow to the application and are continuously evaluated by static queries. However, with IBM InfoSphere Streams, those continuous queries can be modified over time as requirements change.

In addition, as new jobs are submitted, the IBM InfoSphere Streams scheduler determines how it might reorganize the system to best meet the requirements of both newly submitted and already executing specifications, and the Job Manager automatically effects the required changes. The runtime environment continually monitors and adapts to the state and utilization of its computing resources, as well as the information needs expressed by the users and the availability of data to meet those needs.

Results that come from the running applications are acted upon by processes (such as web servers) that run external to IBM InfoSphere Streams. For example, an application might use TCP connections to receive an ongoing stream of data to visualize on a map, or it might alert an administrator to anomalous or other interesting events.

IBM InfoSphere Streams takes a fundamentally different approach for continuous processing and differentiates itself with its distributed runtime platform, programming model, and tools for developing continuous processing applications. The data streams consumed by IBM InfoSphere Streams can originate from sensors, cameras, news feeds, stock tickers, or a variety of other sources, including traditional databases.

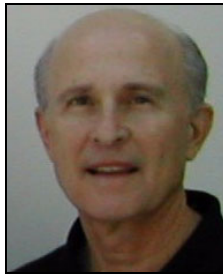
In IBM InfoSphere Streams, continuous applications are composed of individual Operators that interconnect and operate on multiple data streams. Those data streams can come from outside the system or be produced internally as part of an application.

IBM InfoSphere Streams provides an execution platform and services for user-developed applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous streams of data. Developing applications in this type of environment is significantly different because of the continuous flow nature of the data. In the remainder of this book, we describe the environment for this type of processing, provide an overview of the architecture of the IBM InfoSphere Streams offering, and discuss best practices for programming applications to operate in this type of environment.

The team who wrote this book

This book was produced by a team of specialists from around the world working with the International Technical Support Organization, in San Jose, California.

The team members are depicted below, along with a short biographical sketch of each team member.



Chuck Ballard is a Project Manager at the International Technical Support Organization, San Jose, California. He has over 35 years experience, holding positions in the areas of product engineering, sales, marketing, technical support, and management. His expertise is in the areas of database, data management, data warehousing, business intelligence, and process re-engineering. He has written extensively about these subjects, taught classes, and presented at conferences and seminars worldwide. Chuck has both a Bachelor's and a Master's degree in Industrial Engineering from Purdue University.



Daniel M Farrell is an IBM Certified Professional and Pre-sales Engineer from Denver Colorado. In 1985, Daniel began working with Informix® software products at a national retail and catalog company, a job that would set his direction for the next twenty or so years. He joined IBM as a result of the acquisition of Informix Software. Daniel has a Master's degree in Computer Science from Regis University and is currently pursuing a Ph.D from Clemson in Adult Education and Human Resource Studies. Both Daniel's thesis and dissertation concern research on fourth generation programming languages, business application delivery, and staff development.



Mark Lee is a Senior Consultant within the IBM Software Group. He is an experienced field practitioner who has worked with a wide range of IBM customers in parallel processing information systems and service-oriented architectures over a period of 20 years. Mark has an Honors degree and a Master's degree in Computer Science from the University of Manchester. He is located in Surrey, England.



Paul D Stone is an IT Architect with the IBM Emerging Technology Services organization. He has over 20 years of experience in developing IT systems, providing support in software development, architecture, and design, and specializing in performance optimization. Paul has recently developed proof of concept applications for customers on the IBM InfoSphere Streams early adopters' program. He is located at IBM Hursley Park in Winchester, United Kingdom.



Scott Thibault is the founder and president of Green Mountain Computing Systems, Inc. Scott has 15 years of experience with Electronic Design Automation and hardware acceleration. Programming languages and compilers are his areas of specialty and he has been involved with streams-based programming for the past 7 years. Scott holds a Doctoral degree in Computer Science from the University of Rennes I, France and a Master's degree from the University of Missouri-Rolla.



Sandra Tucker is an Executive IT Specialist with the IBM Software Group Information Management Lab Services Center of Excellence who concentrates on services for emerging products and integrated solutions. She has been successfully implementing business intelligence and decision support solutions since 1990. Sandra joined IBM through the Informix Software acquisition in July 2001 and is currently focused on the IBM InfoSphere Streams computing product, IBM InfoSphere Streams, and the Data

Warehousing solutions space. Her experience includes work in health care profile health and cost management, retail sales analysis and customer profiling, and telecommunication churn analysis and market segmentation.

We would like to thank the following people who contributed to this book, in the form of advice, written content, and project support:

Nagui Halim, Director, IBM InfoSphere Streams, Information Management Software Group, Hawthorne, New York

Roger Rea, IBM InfoSphere Streams Product Manager, IBM Information Management Software Group, Raleigh, North Carolina

Warren Pettit, Instructor/Developer for Data Warehouse Education, IBM Information Management Software Group, Nashville, Tennessee

Sam Ellis, Manager, IBM InfoSphere Streams Development, IBM Information Management Software Group, Rochester, Minnesota
IBM USA

Mary Comianos, Publications Management
Ann Lund, Residency Administration
Emma Jacobs, Graphics Support
Wade Wallace, Editor
International Technical Support Organization

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author - all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:
ibm.com/redbooks
- ▶ Send your comments in an email to:
redbooks@us.ibm.com
- ▶ Mail your comments to:
IBM Corporation, International Technical Support Organization
Dept. HYTD Mail Station P099
2455 South Road
Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- ▶ Find us on Facebook:
<http://www.facebook.com/IBMRedbooks>
- ▶ Follow us on Twitter:
<http://twitter.com/ibmredbooks>
- ▶ Look for us on LinkedIn:
<http://www.linkedin.com/groups?home=&gid=2130806>
- ▶ Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:
<https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm>
- ▶ Stay current on recent Redbooks publications with RSS Feeds:
<http://www.redbooks.ibm.com/rss.html>



Introduction

In this chapter, we give you some background about an exciting analytic paradigm called *stream computing*. We introduce you to the concepts of streaming data and stream computing from the business landscape and informational environment perspectives, and the emergence of real-time analytic processing (RTAP). We then build on this fundamental information to show how IBM has been actively involved in this extreme shift in the process of decision making. The result of years of research into how to meet this challenge can be seen in the IBM InfoSphere Streams (Streams) product. To help position this shift, we give you some history about the birth of Streams and how it is designed to provide an optimal platform for developing and deploying key applications to use streaming data in the decisions you need to make, and the actions you need to take for the health of your business and even the quality of your life.

Another of the objectives in this chapter is to provide insight into the following key questions and concepts.

- ▶ What exactly is streaming data?
- ▶ What sort of things can be learned from streaming data to improve the fabric of our lives and the functions of our businesses?
- ▶ How can developing Streams Applications enable you to become more proactive in the decision making process?
- ▶ How does IBM InfoSphere Streams enable you to harness the power of the plethora of information available from the traditional and non-traditional sources of data that are *moving* all around us?

1.1 Stream computing

We live in an age where the functions of communication that were once completely the purview of land lines and home computers are now primarily being handled by intelligent cellular phones, and where digital imaging has replaced bulky X-ray films and the popularity of MP3 players and eBooks are changing the dynamics of something as familiar as the local library.

For years we have listened to self-professed information evangelists take every opportunity to expound on the explosive increase in the amount of available data and the challenge to glean key useful information from the deluge in time to take action. Actually, we have heard these proclamations so often we have become somewhat desensitized to the true impact of such a statement. At a time when this rate of increase is surpassing everyone's projections, many of us still feel the challenge of being able to do something with this tidal wave of information is only relevant within the constructs of our jobs. Unfortunately, this is not nearly the case. The time has come when the flood of data is so linked to all aspects of our lives that we can no longer treat all that information as so much background noise. It is time to be more than just aware that all that information is there; we need to integrate it into our businesses, our governments, our education, and our lives. We need to use this information to make things happen, not just document and review what has happened.

The desire to use this information to make things happen is not a new desire. We have always wanted to use available information to help us make more informed decisions and take appropriate action in real time. The realization of true real-time analysis has been challenged by many things over the years. There have been limitations in the capabilities of the IT landscape to support delivering large volumes of data for analysis in real time, such as processing power, available computing memory, network communication bandwidth, and storage performance. Although some of these limitations still exist, there have been significant strides in minimizing or alleviating some or all of them. Still one of the biggest challenges to presenting data in real time is not directly related to the capabilities of the data center. One of the main challenges has been the ability to acquire the actual information in a way that makes it available to the analysis process and tools in time to take appropriate action.

Even today, typical analytical processes and tools are limited to using stored and usually structured data. Data acquisition has historically required several time-consuming steps, such as collection of the data via data entry or optical scanning, cleansing, transforming, and enriching the data, and finally loading the into an appropriate data store. The time it takes for these steps results in a delay before data is available to be analyzed and drive any actionable decisions. Often,

this delay is enough that any action taken from the analysis is more reactive than proactive in nature.

If we have experienced challenges before in acquiring information in real time, we can only expect this situation to get worse. Our world is becoming more and more instrumented. Because of this phenomenon, traditionally unintelligent devices are now a source of intelligent information. Tiny processors, many with more processing power than the desktop computers of years ago, are being infused into the everyday objects of our lives. Everything from the packages of products you buy, to the appliances in our homes, to the cars we drive now have the capability to provide us with information we could use to make more informed decisions. An example of this situation is shown in Figure 1-1.



Figure 1-1 Information is available from formerly inanimate sources

Along with the proliferation of instrumentation in everyday products and processes, we are experiencing a surge in interconnectivity as well. This means that not only is the actual data available right from the source, but those sources are also interconnected in such a way that we can acquire that data as it is being generated. The acquisition of data is no longer limited to the realm of passive observation and manual recording. The information produced and communicated by this massive wave of instrumentation and interconnectivity makes it possible to capture what is actually happening at the time it happens. Data can be acquired at the source and made available for analysis in the time it takes a machine to *talk* to another machine. Where we once assumed, estimated, and predicted, we now have the ability to know.

These advances in availability and functionality are the driving force in the unprecedented, veritable explosion in the amount of data available for analysis. The question now becomes, are we really ready to use this information? Even with the recent improvements in information technology, the predicted steady

increase in the amount of traditional stored data available was a considerable challenge to realizing true real-time analytics. The world is now able to generate inconceivable amounts of data. This unexpected increase in volume is likely to still outstrip the potential gains we have made in technology, unless we begin to change the way we approach analysis. Just getting that amount of data stored for analysis could prove to be an insurmountable task.

Imagine: In just the next few years, IP traffic is expected to total more than half a zettabyte, that is, a trillion gigabytes.

A further challenge is new sources of data generation. The nature of information today is different than information in the past. Because of the profusion of sensors, microphones, cameras, and medical and image scanners in our lives, the data generated from these items will shortly make up the largest segment of all information available. One advantage of that shift is that these non-traditional data sources are often available while they are enroute from their source to their final destination for storage. This means they are basically available the moment they happen, before they are stored.

For a minute, let us consider data feeds from local or global news agencies or stock markets. Throughout their active hours of operations, these data feeds are updated and communicate data as events happen. Many of us have at least one of the addresses (URLs) for these live data feeds bookmarked on our browser. We are confident that every time we access this address we will be able to review the most current data. On the other hand, we probably do not have any idea where the historic information goes to finally be stored. More importantly, there is really no need to know where it is stored, because we are able to analyze the data before it is stored. This data that is continuously flowing across interconnected communication channels is considered streaming data or *data in motion*.

To be able to automate and incorporate streaming data into our decision-making process, we need to use a new paradigm in programming called *stream computing*. In traditional computing, we access relatively static information to answer our evolving and dynamic analytic questions. With stream computing, you can deploy a static application that will continuously apply an analysis to an ever changing stream of data. In other words, this *continuous analysis* is a static question applied to the ever changing data. In traditional computing, ever changing questions are often applied to fairly static data (data at rest). In stream computing, fairly static questions are continuously being evaluated with new real-time data (data in motion).

Figure 1-2 shows the difference between doing research for a report on a current event using the printed material in a library versus using online news feeds.



Figure 1-2 Comparing traditional and stream computing

To help you better understand this concept, consider for the moment that you are standing on the bank of a rapidly flowing river into which you are planning to launch a canoe. You have a wet suit but are not sure if you need to bother with putting it on. You first need to determine how cold the water is to decide whether or not you should put on the protective gear before getting into the canoe. There is certainly no chance of capturing all of the water and holding it while you determine the average temperature of the full *set* of the water. In addition, you want to leave your canoe while you go to retrieve a newspaper that might have a listing of the local water temperatures. Neither of those options are practical.

A simpler option is to stand on the edge of the creek and put your hand or foot into the water to decide whether it is warm enough to forego the protective gear. In fact, you could also insert a thermometer for a more accurate reading, but that level of accuracy is probably not necessary for this scenario. The water's journey was not interrupted by our need to analyze the information. On the contrary someone else could be testing those same waters farther downstream for their own informational needs. Much like its watery counterpart, streaming data can be accessed from its edges during its travels without deterring further use downstream or stopping its flow to its final repository.

Looking at the different programming approaches to analysis, you can see correlations to the river, although historically you have been able to capture all of the information prior to analysis. In traditional computing, you could produce a report, dashboard, or graph showing information from a data warehouse for all sales for the eastern region by store. You would take notice of particular items of interest or concern in your high level analysis and drill down into the underlying details to understand the situation more clearly. The data warehouse supplying information to these reports and queries is probably being loaded on a set schedule, for example, weekly, daily, or hourly, and not modified much after the data is committed. So the data is growing over time, but historical data is mostly

static. If you run reports and queries today, tomorrow, or a month from now on the same parameters, the results will basically be the same. On the other hand, the analysis in this example goes from summary to detail. You could take action based on this analysis. For example, you could use it to project replenishment and merchandising needs. But because the analyst must wait until the data is loaded, to use it will limit their ability to take those actions in real time. With some analyses and actions, this is a perfectly acceptable model, for example, making plans for re-tooling a production process or designing a new line based on the success of another line. With stream computing, you can focus on an analysis that results in immediate action. You could deploy an analytic application that will continuously check the changing inventory information based on the feeds from the cash registers and the inventory control system and send an alert to the stock room to replenish item(s) more quickly before going through the effort of loading the data.

1.1.1 Business landscape

As previously stated, data acquisition for real-time analysis is becoming more and more of a reality. However, most of our decisions, either as business leaders or even as individuals, are made based on information that is historic in nature or limited in scope. Even though most decision makers believe that more current data leads to better decisions, stored data has been the only source of data readily available for analysis. This situation has driven expectations about how the analytic process and the tools that support analysis should work. Businesses historically had to wait until data had been committed to storage before they could an analysis. This analysis is then limited to using historical, and usually structured, data to predict the best actions for the future. By being able to acquire actual data in real time, businesses hope to be able to analyze and take action in real time, but they need new products and tools designed to perform these real-time actions.

Figure 1-3 shows the results of a 2009 IBM study of approximately 2400 CIOs worldwide. The source of this data is the IBM Institute for Business Value Global CIO Study 2009.

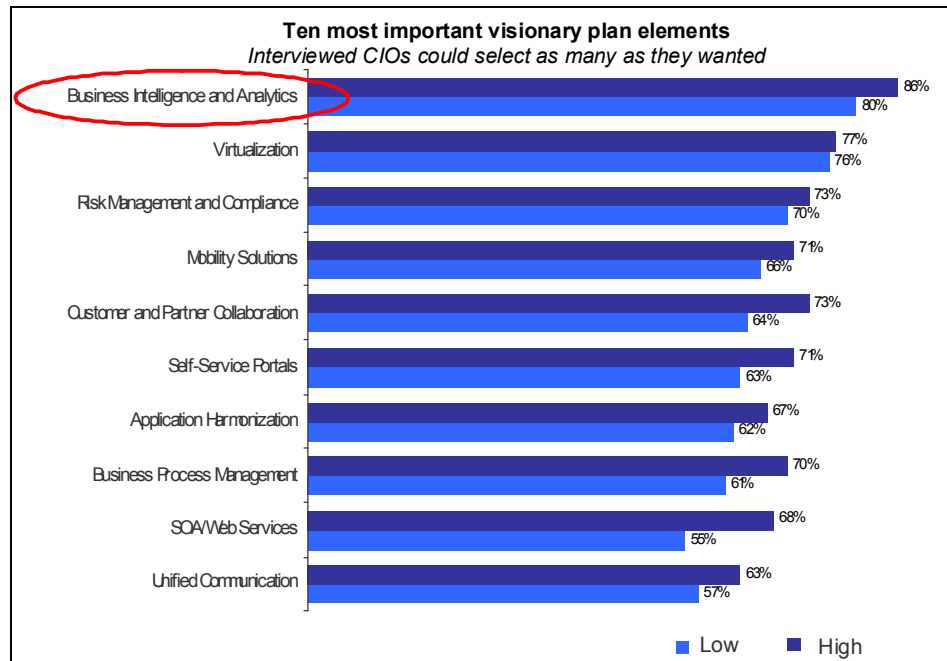


Figure 1-3 Making investments in business intelligence and analytics

These CIOs indicated that investing in the effective use of analytics for decision making remains their top priority. Why do businesses continue to seek an effective use of analytics? Because before you can determine an effective use strategy, the definition of what it takes to be effective changes. The rapid shift in the complexity of the data causes companies, governments, and even individuals to continually try to determine how best to use the new data landscape. An excess of good data might sound like a good thing, but with enterprise data projected to double every 18 months, it is difficult to work with all that data. In addition, with 80% of data growth driven by unstructured and non-traditional data sources, such as email, text, VoIP, and video, it is difficult to even know what types of data for which you will plan. With the fast pace of today's business environment, industry leaders are constantly being called upon to make innovative decisions in real time to gain a competitive edge, or to simply remain competitive.

This is the time when these images and streaming data will be far more prevalent than their historic structured counterparts. Businesses are challenged by the volume and velocity of available data and by their ability to interpret the broad range of sources.

Note: A city the size of London could have tens of thousands of security cameras. Roads are often equipped with thousands of sensors for an area as limited as one bridge. The rise in digital imaging for all types of medical scans will almost guarantee that medical images will comprise almost a third of all the data in the world. Most if not all of these devices are connected to the World Wide Web, where they feed data day and night.

And that list of sources of available data for analysis just keeps growing. The increased reach of instrumentation brings with it the availability of a volume of data that businesses could never have dreamed would exist. We have smart appliances that can keep smart meters aware of energy usage at the lowest level of time and device, which allows the utility company to make capacity adjustments at any level, even down to within a single home. Even something as basic as a tractor can capture and transmit soil condition, temperature, and water content and directly relay that information to a specific location to be used to monitor and manage water usage for irrigation systems, while simultaneously alerting the local dealership of the fact that the machine needs service. From the potential of smart grids, smart rail, smart sewers, and smart buildings, we can see significant shifts in the way decisions are made in the fabric of our lives.

Simultaneously, as individuals we are becoming increasingly aware of the value of real-time information. The rise in popularity of the social networks indicates that there is a considerable personal value to real-time information. In these networks, millions of people (and therefore customers, students, patients, and citizens) are voicing their opinion about everything from good and bad experiences they have had with products or companies to their support for current issues and trends. Businesses that can analyze what is being said and align themselves with popular desires will be more successful.

In some situations, businesses have tried to build applications to provide functionality beyond traditional analytics tools to use non-traditional or high volume information. Unfortunately, many of these businesses find that their custom applications struggle to scale well enough to keep up with the rapidly growing data throughput rates. The reality is that even as the old obstacles to real-time analytics are removed, business leaders are still finding themselves limited in their ability to make truly real-time decisions. Unless they embrace a new analytical paradigm, the dynamic growth in data volumes will result in increasing the time needed to process data, potentially leading to missed opportunities and lowering, or losing, competitive advantage.

Businesses are keenly aware that knowing how to effectively use all known sources of data in a way that allows future sources to be incorporated smoothly can be the deciding factor that serves to fuel competitive, economic, and environmental advantages in real time. Those companies and individuals that find themselves positioned to be able to use data in motion will certainly find themselves with a clear competitive edge. The time it takes to commit the data to persistent storage can be a critical limitation in highly competitive marketplaces. More and more, we will see that the effective key decisions will need to come from the insights available from both traditional and non-traditional sources.

1.1.2 Information environment

You can already see that the changes in the business landscape are beginning to blur the lines between the information needs of business and the needs of the individual. Data that was once only generated, collected, analyzed, enriched, and stored in corporate data centers is now transmitted by some of the most innocuous devices. Data that was once only attainable by using the power provided by a business's information environment can now be access by devices as personal as intelligent cellular phones and personal data assistants. With more open availability comes the potential for a new breed of analyst and new sources of enriching the data. These new analysts had a limited voice until these changes in the information environment occurred. Thanks to the increases in instrumentation and interconnectivity, any voice or signal can now be heard worldwide in the matter of the time it takes to press a key. The lines that historically have delineated the information environment into areas used by industries, business, government, or individuals are becoming all but invisible. The world becoming more instrumented and interconnected, and as a result, the entire planet is becoming more intelligent.

We are at the dawn of the Smarter Planet™ Era of the Information Age. As you can see in Figure 1-4, the Smarter Planet Era builds off earlier styles of computing, and is being driven by the rapid rise in the level of sophisticated computing technology being delivered into hands of the real world.

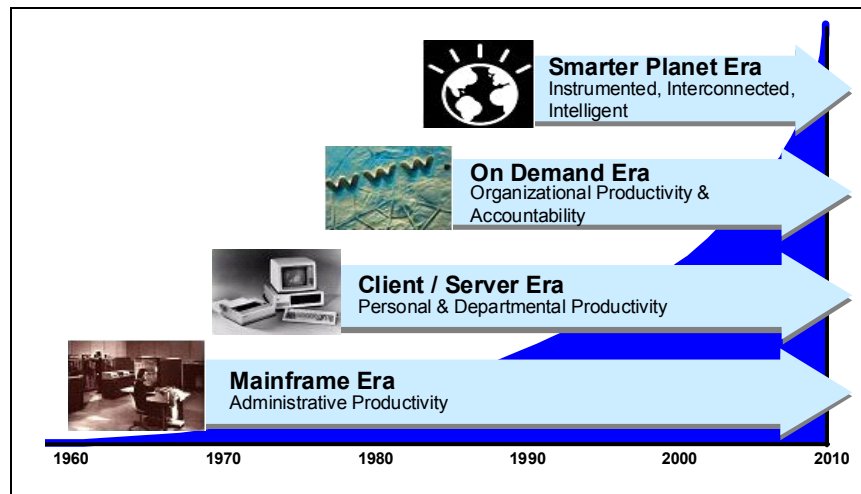


Figure 1-4 Eras of information technology evolution

The need for an automated information environment began in the mid 1960s with the Mainframe Era. This wonderful environment for automating back-office functions is still widely in use today. Because all businesses have back offices, the information environment of the Mainframe Era became prolific in all businesses and all industries. The mainframe environment was not designed for such areas as plant floor manufacturing, departmental computing, or personal computing environment; the data it creates and stores is solely for usage by the specific enterprise. The purpose of the applications in this environment is to conduct day-to-day business. The data it generates is only used by the processes of those day-to-day operations, such as order processing, fulfillment, and billing.

The awareness of the value of analysis rose up from the departmental units within the business. The informational environment of the Mainframe Era was well suited to creating and storing data, but not as suited to analyzing that information to make improvements in the way to do business. As departmental decision makers were pressed to improve their bottom lines, they found themselves needing an information environment flexible enough to support any question they needed to ask. The Client/Server Era was the first information environment fashioned to support the business analyst. Unfortunately, as it rose from departmental demands, it was an information environment isolated to the data needs of the particular departmental unit, a cluster of islands of automation.

Still, it was quick and effective, and like its predecessor, still exists in organizations today for targeted needs, such as campaign management.

But soon analysis could no longer be isolated to the departmental domain. The On Demand Era was born from the need to blend (and validate) the departmental analysis to provide insight and recommendations at the enterprise level. The On Demand Era started integrating these islands of automation into an information environment that could serve the analytical needs of the entire enterprise. The rise of the worldwide internet, with its open, standard-based, and widely available access, provided a framework to share departmental information within the business and the foundation for sharing information throughout the world at a pace we could barely imagine.

You can see that the growth of the environments to support information has been relatively steady over the years, and that the growth has been gradual. The sources of data and the results of analysis were primarily the property of industry, government, and academic science. These enterprise data centers controlled the information because only they had the computing power to acquire and analyze it. As such, they reviewed and verified information before communicating it to the real world. Today, that kind of computing power exists outside of the walls of industry and is prevalent. As a result, we are having to re-think what a computer really is.

Consider: There were more than 4 billion mobile phone subscribers at the end of 2008, and there will be 1.3 billion Radio Frequency Identification tags (RFIDs) produced globally in the next year. Sensors are being embedded across entire ecosystems, supply chains, health care networks, cities, and even natural systems such as rivers. All of these environments have the ability to generate and communicate data.

Once, the information available on the internet was only generated and communicated by those large data centers. Now it is estimated that by as early as 2011 not only will there likely be over 2 billion people connected to the internet, there will also be as many as a trillion objects connected as well. This *Internet of Things*, shown in Figure 1-5, shows some examples of how computers are moving out of the data centers and into our everyday world.

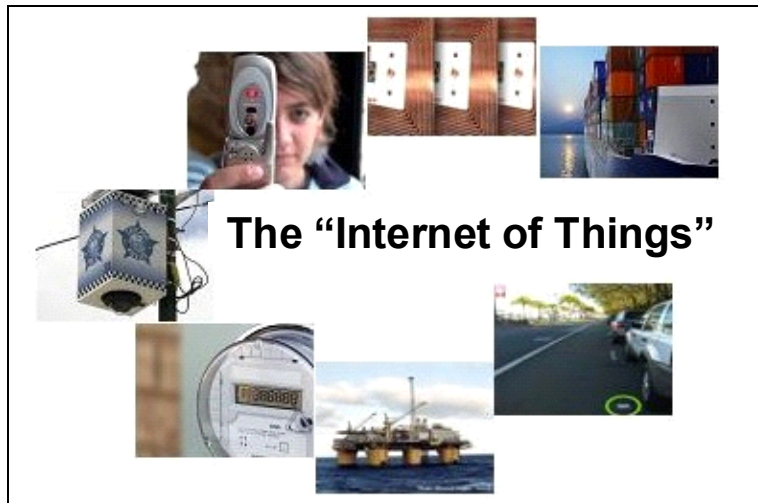


Figure 1-5 Computers are moving out of the data center into the world

The proliferation of these sensors and devices, embedded in everything from smart grids to rail cars, and even chickens, will help drive as much as a tenfold increase in the amount of data in the world. The ability to tap into that data for a new kind of intelligence can help drive vast improvements in our systems.

Welcome to the Smarter Planet Era. Driven by the surge in instrumentation, interconnectivity, and intelligence being infused in all aspects of our lives, this new information environment holds the potential to infuse complex analysis into the routine aspects of our lives. Situations such as having traffic and security cameras in major cities help alert police and other first responders to the logistics of an incident far faster and more precisely than ever before. The information from sensors in your car could alert your mechanic that your car needs maintenance so they can schedule an appointment for you and prompt you to confirm the appointment via a text message on your cell phone. In-home monitors that transmit key health factors could allow an elderly relative to continue living in their own home. Automated quality analysis installed on a factory line could identify defects in products before they leave the line to allow Operators to take corrective measures.

While having access to data sooner might save lives, lower crime, save energy, and reduce the cost of doing business and creating products, one of the by-products of the widespread instrumentation is the potential for feeling like we live in a *surveillance society*. While it is easy for people and businesses to see the value of access to the wide variety of data and smarter systems, it is also easy to see how they might be increasingly uncomfortable having so much information known about them. Individuals and businesses alike find themselves concerned that this essential environment might only be as secure and reliable as the average person's mobile computer or PDA.

In addition to the sheer mechanics of analyzing the massive volumes of information in real time, there is the additional challenge providing security and privacy. The information environment of the Smarter Planet Era gets its reaction speed from accessing data in motion, data outside of the usual framework of data repository authorization and encryption features. The sheer volume of the available data in motion requires a high performance environment for stream computing, but this environment also needs to be able to employ stream analysis to determine the credibility of sources and protect the information without adversely compromising the speed. When the information environment uses and allows wide spread access to analyze and enrich data in motion, it also takes on some of the responsibility to protect the results of that analysis and enrichment of the data and determine that the analysis is based on credible and trusted sources.

The potential of the information environment for this new Smarter Planet Era is great, but so are its challenges. Providing an environment capable of delivering valuable and insightful information for real-time analytics without compromising on quality and security is the standard with which we will measure the success of this era.

1.1.3 The evolution of analytics

Just as the business landscape and information environments are evolving, the approaches to effective analysis are at the dawn of a major evolution. While we might be thinking that the wealth of information swarming around us sometimes seems more like a pestilence than prosperity, the fact is that we now have the capability to glean value from the chaos. Advanced software analytic tools and sophisticated mathematical models can help us identify patterns, correlations of events, and outliers. With these new tools, we can begin to anticipate, forecast, predict, and make changes in our systems with more clarity and confidence than ever before. We stand on the brink of the next generation of intelligence: analysis of insightful and relevant information in real time. This is the real value of a Smarter Planet.

The evolution of the analytic process is shown in Figure 1-6.

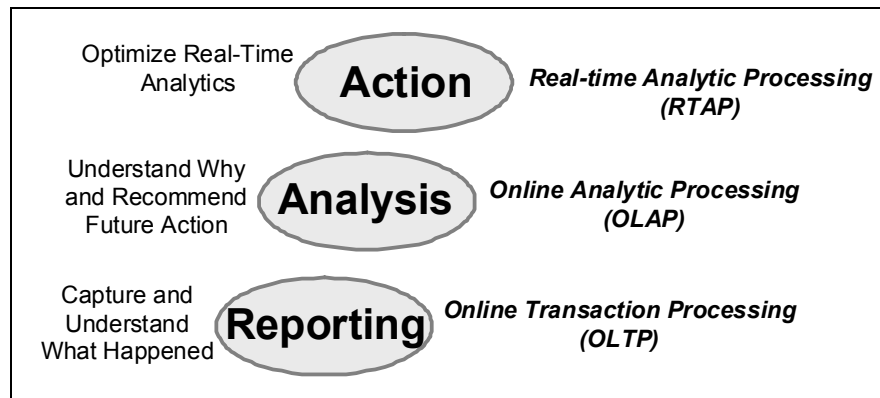


Figure 1-6 The next generation of business intelligence

Hierarchical databases were invented in the 1960s and still serve as the foundation for Online Transaction Processing (OLTP) systems for all forms of business and government driving trillions of transactions. Consider the example of a bank. In many banks, the information is entered into an OLTP system by employees or by a web application that captures and stores that data in hierarchical databases. This information then appears in daily reports and graphical dashboards to demonstrate the current state of the business and to enable and support appropriate actions. Analytical processing here is limited to capturing and understanding what has happened.

Relational databases brought with them the concept of *data warehousing*, which extended the use of databases from OLTP to Online Analytic Processing (OLAP). Using an apparel manufacturer as an example, the order information that is captured by the OLTP system is stored over time and made available to the various business analysts in the organization. With OLAP, the analysts could now use the stored data to determine trends in fulfillment rates, patterns of inventory control issues, causes of order delays, and so on. By combining and enriching the data with the results of their analyses, they could do even more complex analyses to forecast future manufacturing levels or make recommendations to accelerate shipping methods. Additionally, they could mine the data and look for patterns to help them be more proactive in predicting potential future problems in areas such as order fulfillment. They could then analyze the recommendations to decide if they should take action. Thus, the core value of OLAP is understanding why things happen, which allows you to make more informed recommendations.

A key component of both OLTP and OLAP is that the data is stored. Some new applications require faster analytics than is possible, and you have to wait until the data is retrieved from storage. To meet the needs of these new dynamic applications, you must take advantage of the increase in the availability of data prior to storage, that is, streaming data. This need is driving the next evolution in analytic processing, Real-time Analytic Processing (RTAP). RTAP focuses on taking the proven analytics established in OLAP to the next level. Data in motion and unstructured data might be able to provide actual data where OLAP had to settle for assumptions and hunches. The speed of RTAP allows for taking immediate action instead of simply making recommendations.

So, what type of analysis makes sense to do in real time? Key types of RTAP include, but are not limited to:

► Alerting

- The RTAP application notifies the user(s) that the analysis has identified that a situation (based on a set of rules or process models) has occurred and provides options and recommendations for appropriate actions.
- Alerts are useful in situations where the application should not be automatically modifying the process or automatically taking action. They are also effective in situations where the action to be taken is outside the scope of influence of the RTAP application.

Examples:

- A market surveillance application that is monitoring a local exchange for suspect activity would notify someone when such a situation occurs.
- A patient monitoring application would alert a nurse to take a particular action, such as administering additional medicine.

► Feedback

- The RTAP application identifies that a situation (based on a set of rules or process models) has occurred and makes the appropriate modifications to the processes to prevent further problems or to correct the problems that have already occurred.
- Feedback analysis is useful in, for example, manufacturing scenarios where the application has determined that defective items have been produced and takes action to modify components to prevent further defects.

Example: A manufacturer of plastic containers might run an application that uses the data from sensors on the production line to check the quality of the items through the manufacturing cycle. If defective items are sensed, the application generates instructions to the blending devices to adjust the ingredients to prevent further defects from occurring.

- ▶ Detecting failures
 - The RTAP application is designed to notice when a data source does not respond or generate data in a prescribed period of time.
 - Failure detection is useful in determining system failure in remote locations or problems in communication networks.

Examples:

- ▶ An administrator for a critical communications network deploys an application to continuously test that the network is delivering an adequate response time. When the application determines that the speed drops below a certain level or is not responding at all, it alerts the administrator.
- ▶ An in-home health monitoring application determines that the motion sensors for a particular subscriber have not been activated today and sends an alert to a caregiver to check on the patient to determine why they are not moving around.

When you look at the big picture, you can see that if you consider the improvements that have been made in chip capacity, network protocols, and input / output caching along with the advances in instrumentation and interconnectivity and the potential of stream computing, we stand poised and ready to be able to present appropriate information to the decision makers in time for proactive action to be taken. The days of solely basing our decisions on static data is yielding to being able to also use streaming data or *data in motion*.

1.2 IBM InfoSphere Streams

In May of 2010, IBM made a revolutionary product, IBM InfoSphere Streams (Streams), available for clients. Streams is a product architected specifically to help clients continuously analyze massive volumes of streaming data at extreme speeds to improve business insight and decision making. Based on ongoing, ground-breaking work from an IBM Research research team, Streams is one of the first products designed specifically for the new business, informational, and analytical needs of the Smarter Planet Era.

As shown in Figure 1-7, Streams is installed in over 90 sites across six continents. The wide variety of use cases encompassed by these installations is helping IBM understand more about client requirements for this new type of analysis.

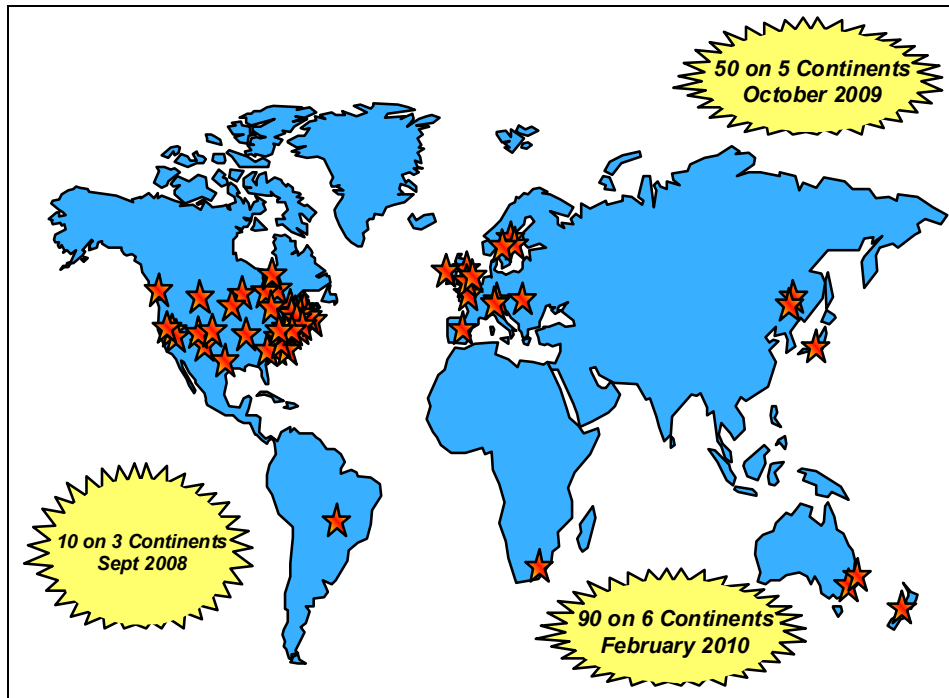


Figure 1-7 Streams installations as of February 2010

Streams is based on nearly a decade of effort by IBM Research to extend computing technology to handle advanced analysis on high volumes of data quickly. How important is this research? Consider how it would help crime investigation to be able to analyze the output of any video cameras in the area surrounding the scene of a crime to identify specific faces for persons of interest in the crowd and relay that information to the unit that is responding. Similarly, what a competitive edge it could be to be able to analyze 5 million stock market messages per second and execute trades with average latency of under 100 microseconds (faster than a hummingbird flaps his wings). Think about how much time, money, and resources could be saved by analyzing test results from chip manufacturing wafer testers in real time to determine if there are defective chips before they leave the line.

Note: While at IBM, Dr. Ted Codd invented the relational database. It was known as *System R* while it was at IBM Research. The relational database is the foundation for data warehousing, which launched the highly successful Client/Server and On Demand Eras. One of the cornerstones of that success was the capability of OLAP products, which are still used in critical business processes today.

When IBM Research again set its sights of developing something to address the next evolution of analysis (real-time analytic processing (RTAP)) for the Smarter Planet Era, they set their sights on developing a platform for the same level of world changing success and decided to call their system *System S*. Like System R, System S was founded on the promise of a revolutionary change to the analytic paradigm. The research of the Exploratory Stream Processing Systems team at T.J. Watson Research Center, which works on advanced topics in highly scalable stream processing applications for the System S project, is the heart and soul of Streams.

Critical intelligence, informed actions, and operational efficiencies that are all available in real time is the promise of Streams, and Streams is what will help us realize the promise of a Smarter Planet.

1.2.1 Overview

As the amount of data available to enterprises and other organizations dramatically increases, more and more companies are looking to turn this data into actionable information and intelligence in real time. Addressing these requirements requires applications that are able to analyze potentially enormous volumes and varieties of continuous data streams to provide decision makers with critical information almost instantaneously. Streams provides a development platform and runtime environment, where you can develop applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams based on your defined proven analytical rules and alert you to appropriate action in time to take that action.

Streams goes further by allowing the applications to be modified over time. While there are other systems that embrace the stream computing paradigm, Streams takes a fundamentally different approach to continuous processing and differentiates itself from the rest with its distributed runtime platform, programming model, and tools for developing continuous processing applications. The data streams consumable by Streams can originate from sensors, cameras, news feeds, stock tickers, or a variety of other sources, including traditional databases. Streams of input sources are defined, and can be numeric, text, or non-relational types of information, such as video, audio, sonar,

or radar inputs. Analytic Operators are specified to perform their actions on the streams.

A simple view of this distinction is shown in Figure 1-8.

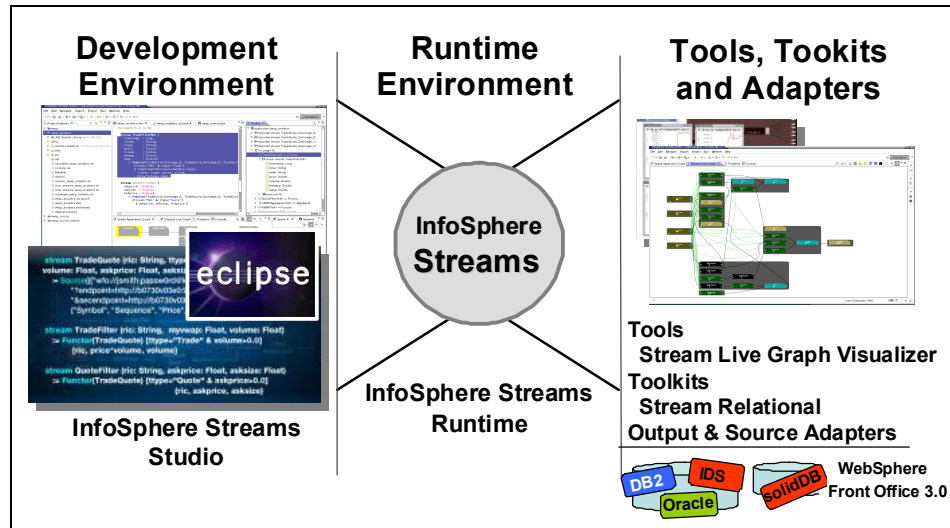


Figure 1-8 The components of IBM InfoSphere Streams

Applications are developed in Streams Studio using Streams Processing Language (previously called Streams Processing Application Declarative Engine), a declarative language customized for stream computing. Once developed, the applications are deployed to the Streams Runtime environment. Streams Live Graph then enables you to monitor performance of the runtime cluster, both from the perspective of individual machines and the communications between them.

Virtually any device, sensor, or application system could be defined using this language. But there are also predefined source and output adapters that can further simplify application development. As examples, IBM delivers the following adapters:

- ▶ TCP/IP, UDP/IP, and files.
- ▶ IBM WebSphere® Front Office, which delivers stock feeds from major exchanges worldwide.
- ▶ IBM solidDB® includes an in-memory, persistent database using the Solid® Accelerator API.
- ▶ Relational databases, supported by using the industry standard ODBC.

Applications, such as the one shown in the application graph in Figure 1-9, often have multiple steps. For example, say you need to calculate volume weighted average prices for stocks on the New York Stock Exchange (NYSE). Your application would source the NYSE data and calculate volume weighted average prices (VWAP). To expand this application into actionable analytics, you could include and parse the quarterly text reports from the US Securities and Exchange Commission (SEC) Edgar database to find quarterly earnings and other information, merge data with the calculated volume weighted average price, perform some custom analytic rules for what you deem to be a good value, and provide alerts about a buy or sell decision or allow the application to decide whether or not to initiate the buy or sell.

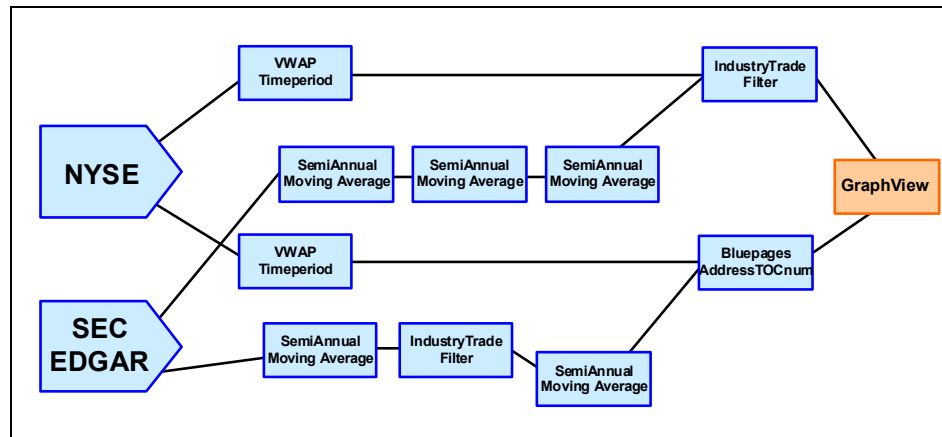


Figure 1-9 Application graph of a Streams Application

Streams delivers an Integrated Development Environment (IDE) based on the Open Source Eclipse project, as shown in Figure 1-10 on page 21. Developers can specify an Operator to be used in processing the stream, such as filter, aggregate, merge, transform, or much more complex mathematical functions, such as fast fourier transforms. The developer also specifies the data streams to be used (Source) and created (Sink). Some Source and Sink adapters come with the Streams product, but the environment allows the developer to define custom adapters and even custom analytics or Operators in C++ or Java™. Existing analytics can also be called from Streams Applications. More details about how this action is accomplished can be found in the subsequent chapters of this book.

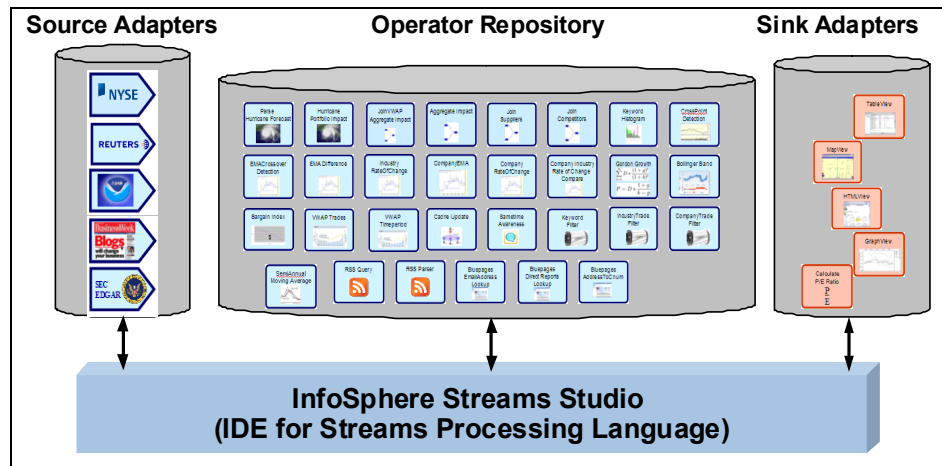


Figure 1-10 The Streams Integrated Development Environment (IDE)

Streams Studio not only helps you develop applications but includes the powerful Streams Debugger tool. As with most IDEs, you can set breakpoints and stops in your application to facilitate testing and debugging. In addition, Streams Debugger works with Streams Runtime and the Streams Live Graph tool so that you can find the breakpoint even after the application is deployed. You can inspect or inject new data to ensure the application is doing what you want it to do.

An optimizing compiler translates the application to executable code. When deployed, the Streams Runtime works with the optimizing compiler to allocate the application across the runtime configuration, as shown in Figure 1-11.

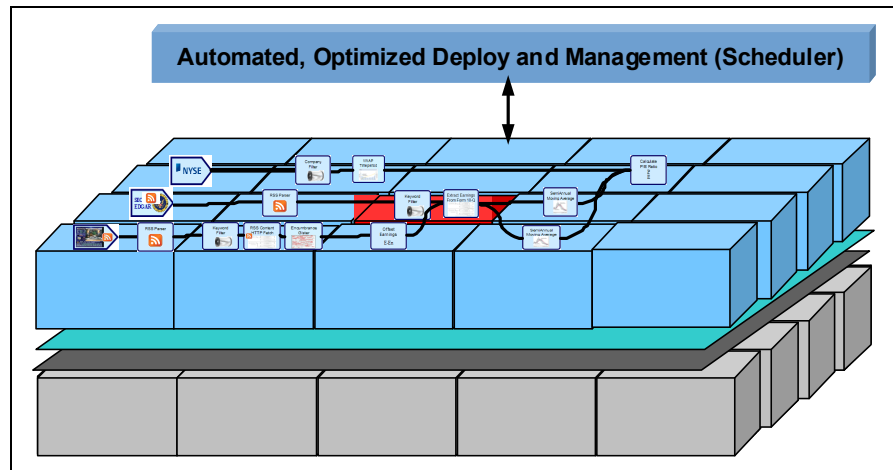


Figure 1-11 An illustration of a typical runtime deployment of a Streams Application

The Streams Runtime establishes communications for the streams of data as they are being processed by the application by setting up connections to route input streams, intermediate streams, and output streams through the cluster. Because the Streams Runtime is all handled in memory across systems within a high speed infrastructure, the overall application experiences extremely low latency for communications.

Streams Runtime monitors the environment's performance and if it detects poor performance, or hardware failure, the administrator can take corrective actions, such as moving an analytic function to another processor in the cluster. The Streams Runtime allows you to add or remove applications without bringing down the cluster.

For the purposes of this overview, it is not necessary to understand the specifics, as they will be discussed in more detail in subsequent chapters. Our purpose here is only to demonstrate how Streams was designed and architected to focus on the ability to deliver real-time analytic processing on exceedingly large volumes of data using a flexible platform positioned to grow with the increasing needs of a dynamic market.

1.2.2 Why IBM InfoSphere Streams

The architecture of IBM InfoSphere Streams represents a significant change in the organization and capabilities of a computing platform. Typically, stream computing platforms are designed with the lofty goal of achieving success toward the following objectives:

- ▶ A parallel and high performance stream processing software platform capable of scaling over a range of hardware capabilities.
- ▶ An agile and automated reconfiguration in response to changing user objectives, available data, and the intrinsic variability of system resource availability.
- ▶ Incremental tasking in the face of rapidly changing data forms and types.
- ▶ A secure, privacy-compliant, and auditable execution environment.

While there have been academic and commercial initiatives focused on solving the above technical challenges in isolation, Streams attempts to simultaneously address all of them. Streams is on a mission to break through a number of fundamental barriers, to enable the creation of solutions designed to deliver on all of these critical objectives.

To do so, the Streams development environment provides a graphical representation of the composition of new applications to make them easier to understand. New applications, or new rules for analysis, can be created dynamically, mapped to a variety hardware configurations, and deployed from the development environment for flexibility and agility of administration. As analytical needs or rules change or priorities and the value of data sources shift, the affected portions of the applications can be modified and re-deployed incrementally with minimal impact to the rest of the application. As hardware resource availability grows within the data center, Streams is designed to be able to scale from a single node to high performance clusters with over a hundred processing nodes. The runtime component continually monitors and adapts to the state and utilization of its computing resources and the data.

There have been other products engineered to focus on some of the above objectives, many of which center around the analysis of business events. Analyzing business event processing can dramatically improve the way a business operates. You should think of an event as something that happens, such as a temperature reading, a purchase, or an insurance claim, while a data stream is somewhat different. Streaming data is more of a continuous reading, such as an EKG, the output of a video camera, or the sound captured by a microphone. Data streams can be viewed as a continuous stream of events happening quickly, and as such there are often correlations and comparisons between stream computing and business event processing.

IBM has been aware of the potential of using business event processing for quite some time. To get a better idea of exactly how to use this dynamic, IBM has classified seven major opportunities for business event analysis and determined the likely corresponding entry points:

- ▶ Business Activity Monitoring
Communicating key performance indicators (KPIs) and alerts to business stakeholders
- ▶ Information-Derived Events
Identifying correlating events across data stores and data streams
- ▶ Business Logic Derived Events
Identifying correlating events across transactions and business processes
- ▶ Active Diagnostics
Identifying potential issues and taking action
- ▶ Predictive Processing
Identifying future issues based upon business events in context and over time
- ▶ Business Service Management
Ensuring that IT health so service level agreements are met
- ▶ High Volume Stream Analytics
Analysis of high volume data streams in real time

Complex Event Processing (CEP) is an evolution of business event processing. It is primarily a concept that deals with the task of processing multiple business events with the goal of identifying the meaningful characteristics within the entire scenario. In addition, Business Event Processing (BEP) has been defined by IBM and handles both simple and complex events in a business context.

The goal is always to be able to take advantage of an appropriate action plan in real time. Because of that goal, CEP tools often have the same, or similar, marketing messages as IBM InfoSphere Streams. However, the proof points behind those messages are typically quite different. At the core, these goals of ultra-low latency and event stream processing in real time are true for both. Although CEP tools usually achieve response times of under a millisecond, Streams has achieved response times in the sub-hundreds of microsecond range. Both types of tools are able to use parallel processing platforms and techniques, such as partitioning and pipelining, to achieve performance, but the extent of that achievement is far from the same.

Streams has some similarity to CEP tools, but it is built to support higher data volumes and a broader spectrum of input data sources. Streams can successfully process millions of messages per second, while CEP tools typically process hundreds of thousands of messages per second. While CEP tools are able to handle discrete events, Streams can handle both discrete events and real-time data streams, such as video and audio. As we have previously described, it also provides an infrastructure to support the need for scalability and dynamic adaptability, using scheduling, load balancing, and high availability, which will certainly be key components as information environments continue to evolve.

Like Streams, CEP tools and products use techniques such as detection of complex patterns of many events, event correlation and abstraction, event hierarchies, and relationships between events to identify *events of events*. These complex events allow for analysis across all the layers of information to determine their impact. CEP tools often employ *if/then/else* type rules-based analysis, but with Streams, you are able to incorporate powerful analytics, such as signals processing, regressions, clustering, and more.

In Table 1-1, we list a few characteristics of Streams and CEP to help clarify and differentiate them.

Table 1-1 Characteristics comparison of Streams and CEP tools

Complex Event Processing	IBM InfoSphere Streams
Analysis of discrete business events.	Analytics of continuous data streams.
Rules based (if/then/else) with correlation across event types.	Supports simple to extremely complex analytics and scales for computational intensity.
Only structured data types are supported.	Supports a whole range of relational and non-relational data types.
Modest data rates.	Extreme data rates.

As you can see, Streams is fully aligned with the challenges of the Smarter Planet Era. The optimized runtime compiler is architected to manage the extremely high volume of data that comprises the current and future business landscapes with the low latency required by real-time analytical processing. The development environment and toolkits are focused on leveraging a client's existing, as well as creating new and complex, analytics to find credibility and value for multiple, and even the non-traditional, information sources that are becoming key elements in our ever broadening information environment.

1.2.3 Example Streams implementations

Even throughout its research and development phase, Streams has been demonstrating success by delivering cutting edge commercial and scientific applications. Many of these applications have shown clients the way to capture a formerly unattainable competitive edge, but in addition, many of these early applications are playing a significant role in the Smarter Planet evolution.

From the beginning, the intent was for the Streams infrastructure to provide a solid foundation for a broad spectrum of practical applications by being designed specifically to address the fundamental challenges of analysis on streaming data. By being able to continually deliver exceptional performance for all types of analyses handling massive amounts of data while remaining flexible enough to offer interoperability with the existing application infrastructures, Streams is positioned to be considered for any solution that would benefit from analyzing streaming data in real time. The uses are as varied as the imagination of the analysts involved in almost every industry and initiative.

Some of the early adopters of Streams have developed solutions with life changing potential. To demonstrate some of the types of things you can do with Streams, we provide, in the following sections, brief overviews of a few of the solutions that have been, or are being, developed.

Radio astronomy: Capturing information about the universe

As previously mentioned, one of the strengths of Streams is the ability to identify items that merit deeper investigation from analytics on data-intensive streams. An example of this scenario is in the area of astronomy, where there are several projects underway worldwide that are using continuously streaming telemetry data from radio telescopes. This multi-national and multi-scientific discipline effort hopes to create the first fully digital radio observatory for astrophysics, space and earth sciences, and radio research. This effort is relying on a Low Frequency Array (LOFAR) of smaller and therefore less expensive telescopes. LOFAR deals with streams of 3D electric fields data containing cosmic ray showers and gamma ray bursts.

These telescopes can have upward of thousands of antennae that are capturing and transmitting data that must be combined to provide the complete set of information about a location in the universe. The challenge is that the data also has a low data usefulness, or information density. For example, it is normal to expect as many as 10 gamma rays per year, each of which will be no more than a few nanoseconds in duration. But to capture those observations requires monitoring all of the messages for the radar installations, which are growing to 72 million message per second containing approximately 500 bytes per messages for about 37 terabytes of data per second.

A Streams Application plays a key role in handling the challenge of correlating this data and identifying what is of interest by providing a more flexible approach to processing this torrent of data. The application was developed for the LOFAR Outrigger in Scandinavia project (LOIS) by a low frequency radio astronomy university group who developed analytics that identify anomalous and transient behavior such as high energy cosmic ray bursts (Figure 1-12).

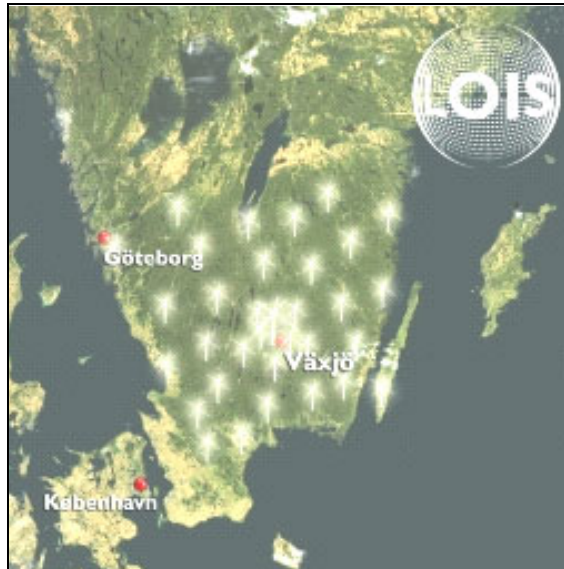


Figure 1-12 Low frequency Array (LOFAR) Outrigger In Scandinavia

There is an initiative to extend LOFAR into a trans-national and cross-disciplinary initiative to build the world's largest and first fully digital radio observatory for astrophysics, space and earth sciences, and radio research. For more information about LOIS, see the following location:

<http://www.lois-space.net/index.html>

Health monitoring: Where decisions can save lives

With the rising costs of health care, stream computing might provide a way to perform better medical analysis without increasing the workload for health care providers. With the ability to include the data generated by the medical devices used to provide constant monitoring of patients both in and out of hospital settings, Streams is a compelling component of solutions targeted at early detection analysis. There is a strong emphasis on data provenance in this domain, that is, in tracking how information is derived as it flows through the application.

In the hospital setting, we have an application that analyzes privacy-protected streams of medical monitoring data directly from the medical devices monitoring an individual patient to detect early signs of disease, correlations among multiple patients, and efficacy of treatments. A first of a kind collaboration between IBM and the University of Ontario Institute of Technology uses Streams to monitor premature babies in a neonatal unit and has demonstrated the ability to predict the onset of critical situations such as sepsis as much as 24 hours earlier than even experienced ICU nurses. Early detection is a key factor in a more positive outcome, shorter hospital stays, and reduced costs for both hospitals and insurers. For more information about this topic, go to the following address:

http://www.uoit.ca/EN/main2/about/news_events/news_archives/news_releases/351026/20091112.html

Another application that is being explored for Streams is in the area of remote assisted living at home, which would enable elderly patients to stay in their own homes longer, thus reducing the cost of their health care without reducing their quality of life. By placing key medical devices and sensors in the home that are used to monitor the patient, the data can be transmitted to an agency that could use Streams to translate, correlate, and analyze the data immediately to provide necessary alerts to a caregiver, doctor, or emergency personnel to provide more direct assistance when needed without requiring the patient to be in a hospital setting.

Manufacturing: Minimizing waste in fabrication

The IBM Burlington and East Fishkill semiconductor chip fabrication lines implemented a prototype of a Streams solution that performs real-time process fault detection and classification using multivariate monitoring. Any process errors that cause defects in manufactured chips can be detected within minutes rather than days or weeks. This quick identification potentially allows the defective wafers to be reworked before subsequent phases of production might render the wafers unusable. Most important though, this quick identification can alert technicians of adjustments that can be made on the line for quality management before more defective wafers are produced¹, as shown in Figure 1-13 on page 29.

¹ <http://portal.acm.org/citation.cfm?id=1376616.1376729>



Figure 1-13 Continuous tuning for quality management

You can easily see how the cost savings can be significant. At a time when businesses are looking for ways to hold the line on expenses, this type of solution could have a broad appeal. For example, a plastic container manufacturer could identify when there are anomalies in the ingredients levels for a product and alert a technician to make adjustments even as products continue down the line. This scenario could be applied to almost every form of manufacturing. Any multi-step process with sensors on the line could potentially analyze the products on the line using established quality measurements that were formerly only available long after the products had left the lines. Any production process, from tires to candy, could be analyzed during the production process in time to make adjustments using either new or the already proven quality metrics. The cost savings is obviously compelling, but the reduction of waste is also a key benefit of such an application.

Smarter cities: We can get there from here

The city of Stockholm, Sweden worked with IBM to implement a system of cameras, lasers, sensors, and transponders to detect all cars entering the city. People are billed if they enter the city at peak hours. After a pilot, a referendum was passed by the citizens to make this system permanent, as it significantly reduced congestion and smog.

A professor at a Stockholm university described this system as a sophisticated batch billing system, in which the data is collected, entered, and stored, and the

bills produced and delivered periodically. This professor is now working to get the entry information in real time so he can merge it with real-time taxi source-destination information, traffic light loop sensor data (to detect congestion), and subway train information to provide the foundation for an Intelligent Transportation system. He used Streams to process the data in motion, and feels this integrated data has the potential to support many real-time analytical applications.

The first of these applications would display, at Arlanda Airport, a real-time map of the city showing the fastest options to reach different parts of the city, depending on factors such as congestion and train departure time. In Figure 1-14, the inset shows a graphical representation of the multitude of data points that are analyzed for a single transportation method. Multiply these data points by the variety of transportation methods and the changing factor of time and you get a feeling for the type of volume Streams can process while maintain performance consistent with real-time analysis.

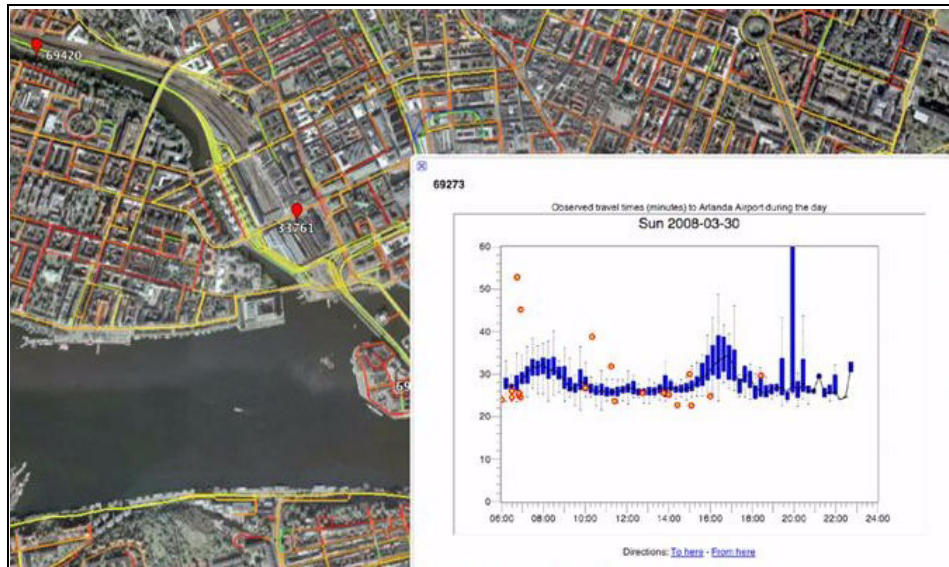


Figure 1-14 Smart Traffic™ Real-Time Traffic Planner

Financial Services: Trading in real time

The financial services industry is under increasing pressure in these tough economic times to find critical success factors to remain competitive. The strength of their business and trading decisions relies heavily on their ability to rapidly analyze increasing volumes of data. Market activity messages must be consumed at rates exceeding millions of messages per second. To make matters even more challenging, that volume is increasing at an alarming rate.

This dramatic growth in market data is expected to continue for the foreseeable future, outpacing the capabilities of many current technologies. Historic data analysis (even the historical data of a few minutes ago) could be the difference between a successful and profitable trading decision and a disastrous one. To remain a leader, or even to remain competitive, this analysis must be real time so the action can be real time.

If the high volume and low latency demands are enough of a challenge, industry leaders are always seeking to extend and refine their strategies by including other types of data in their automated analysis and business awareness. These new sources range from advanced weather prediction models to broadcast news. Streams could define a platform for streaming financial services applications that will continue to scale with industry growth for years.



IBM InfoSphere Streams concepts and terms

In Chapter 1, “Introduction” on page 1, we introduced IBM InfoSphere Streams (Streams) from a business and strategic perspective. In this chapter, we look at the next level of detail by discussing and describing the primary concepts of Streams along with many of the terms typically used in a Streams environment. This will provide a high-level perspective of Streams that will better enable you to understand the lower levels of detail presented in the subsequent chapters.

We have organized the chapter contents into four major sections. First, we discuss the technical challenges found with the types of business problems that Streams is designed to address. However, if you already can easily see where and how Streams can solve those types of problems with which common relational databases and standard extract-transform-load software packages would struggle, you might then choose to advance to 2.2, “Concepts and terms” on page 39, and begin immediately with the ideas that are specific to Streams.

In 2.3, “End-to-end example: Streams and the lost child” on page 49, we detail one sample Streams Application end-to-end, as a means to reinforce the concepts and terms introduced in this chapter, as well a brief introduction to the numerous tools included with Streams.

In 2.4, “IBM InfoSphere Streams tools” on page 64, we introduce Streams administrative and programming interfaces, some of which were used as we created our end-to-end example.

2.1 IBM InfoSphere Streams: Solving new problems

The types of business problems that IBM InfoSphere Streams is designed to address are fundamentally different than those regularly addressed by common relational databases and standard extract-transform-load (ETL) software packages. Consider the following fictitious example scenario.

After scheduling and receiving planned maintenance on your automobile, you pick up your automobile at 7:00 a.m. But then, as soon as you enter the multi-lane highway, proceed to the lane closest to the center of the highway (typically considered the lane for vehicles desiring to travel at higher speeds) and come to speed, your automobile begins running roughly, starts slowing down, and subsequently the engine quits and the automobile comes to a halt.

This is a scenario in which none of us want to find ourselves. It is not only frustrating, but, in this case, extremely dangerous. Even staying in your automobile cannot be considered safe.

So, you decide to exit the automobile and try to get away from the traffic flow. However, the only way you can get off the highway is on foot and by crossing several lanes of high speed traffic (Figure 2-1). But, there are an endless number of high-speed data points to examine to enable decision making. So, how can you do that safely?



Figure 2-1 Examining a stream of traffic

For the moment, let us think of the situation you are experiencing in terms of a data processing problem, that is, how can you find a safe path you can use (time and route) to cross the width of the highway while there are numerous vehicles, travelling at various and high speeds. As is typical of any highway, there are many types of vehicles of different shapes and sizes, that is, some of the oncoming vehicles are longer than others, or are higher and wider, thus obscuring your vision as to what is behind them. How can you make a decision as to when it is safe to cross?

If you did not have a good, continuous view (the live view) of the traffic moving on the highway as it is occurring, would you feel comfortable deciding when to cross the road? Even if you had, say, a satellite photo of the roadway that was 60 seconds old, would you feel safe then? Before you answer, consider that satellite images are static. As such, you would need several satellite images to even try to extrapolate the relative speed of each automobile. And, in the time you are performing these extrapolations the data would have already changed, introducing a measure of latency and resultant inaccuracy into your decision making.

So, you really need is the ability to see a continuous view of the traffic flowing on the highway, with accurate representations of all the vehicles and their properties (length, size, speed, acceleration, and so on). Can you get that with traditional data processing capabilities?

Consider: What if, for example, you were to choose to model the above as an application inside a common relational database server (database):

- ▶ A database could easily model and record how many and the types of vehicles that passed through certain points of the roadway at certain (past) times. The database could easily retrieve this data by location, or most other indices, for as far back as you had chosen to capture and store this data on hard disk (a historical record).
- ▶ If you needed to track something resembling real-time (live) data, you would have a design choice to make: Do you model to track data by vehicle (its current location and speed) or by section of roadway?

Note: In a relational database, the data model is generally static. Even if the model is not static, the data model is not changing every few seconds or sub-second.

While the data in a standard relational database changes, you do not change the whole of the data frequently. Instead you insert, or update a small percentage of the entire contents of the database, where that data resides forever (or at least until it is deleted).

In a relational database, it is the queries that are dynamic. Queries can be *ad hoc* (completely unanticipated), but as long as the answer is to be found inside your database (you have modelled the data to support a given set of questions and program requirements), you will eventually receive a response.

Modeling by vehicle has the advantage that it is finite; there are only so many vehicles that you need to track. If you need to determine conditions for a given section of road, that is, the product of vehicles and vehicle data at a given geographic location, you could create a vehicle record, and update that record on a frequent basis (as frequently as you can afford to from a hardware and performance stand point).

You could also model to track data by location on the roadway, but that is a huge amount of data that you have to update and record as frequently as your hardware can support. Highways and locations that are modelled and recorded inside databases generally have to focus on the relatively small number of known exits and merge points. Everything else is extrapolated,

which does not help you if your automobile is stranded between recorded points.

- A database could easily track the number of maintenance procedures performed on all automobiles by vehicle identifier, by service technician, or by automobile dealer location. The database could then easily answer an unforeseen question, such as how often does a given vehicle identifier return for free service within (n) days of having had some other service, that is, how many service defects and thus disabled automobiles does a given intersection of data produce, as was the case in this actual example.

However, even with all this data, can you satisfy the requirements of the application and get yourself safely off the highway? The short answer is no. You need a new application paradigm to satisfy these new requirements. That new paradigm is available, and is known as stream computing. As we discuss and describe this new paradigm, you will see how using it could satisfy these new types of application requirements.

IBM InfoSphere Streams: A new and dynamic paradigm

As previously stated, a standard database server generally offers static data models and data, while supporting dynamic queries. So, the data is landed (stored) somewhere (most often on a computer hard disk), and is persistent. Queries can arrive at any time and answer any question that is contained inside the historical data and data model.

However, in an IBM InfoSphere Streams environment, you do not have to land data. In fact, with Streams, it is generally expected that the observed data volume can be so large that you could not afford to make it persistent.

In simple terms, Streams supports the ability to build applications that can support the arrival of these huge volumes of data that have the critical requirement to be analyzed, and reported upon, with low latency, and close to or in real time. For that scenario, static models are not sufficient; you need to be able to analyze the data while it is in motion. That is the value of Streams.

Note: The types of questions asked of data stored in a database (landed data) are typically quite different than those asked of real-time data that is in flight.

For example, a standard relational database could easily answer a question about the retail price of every item sold for the past 6 months. Whereas, Streams could answer questions about the current sub-second pricing at a world wide auction per sale item and location. The database is appending new data in a database for subsequent query, whereas Streams is observing a window of live real-time data in flight as it flows from the Source.

Streams operates on (ingests, filters, analyses, and correlates) data as that data is in flight, that is, while the data is still moving. By not having to land data for processing, Streams addresses the types of applications with requirements for high data volumes and low latency responses.

Standard database servers generally have a static data model and data, and dynamic (albeit often long running) queries. IBM InfoSphere Streams supports a widely dynamic data model and data, and the Streams version of a query runs continuously without change. In Figure 2-2, we provide an illustration of these two approaches to data analysis.

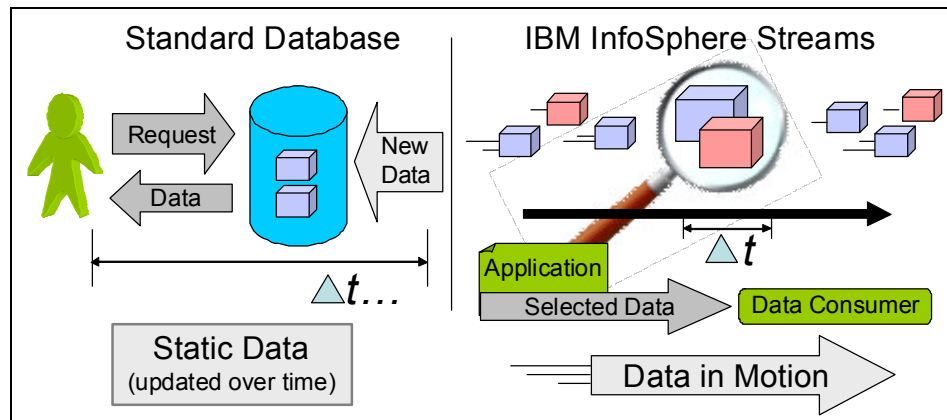


Figure 2-2 Standard relational database compared to IBM InfoSphere Streams

The traditional approach, shown on the left, involves storing data in a standard database, updating that static data over time, and periodically executing queries against that data. By contrast, in the Streams approach shown on the right, the data of interest in the stream is defined, and then processed as it flows by the application, that is, the data is processed while it is in motion and it is typically not stored.

Queries in IBM InfoSphere Streams are defined by the Streams Application, and run continuously (or at least until someone cancels them). In a database environment, a hard disk typically holds the data and the requester is the consumer of that data. In a Streams environment, data meeting the defined criteria is selected as it flows past the application, which then sends it to the appropriate consumer for that data.

Because Streams Applications are always running, and continually sending selected data that meets the defined criteria to the consumers, they are well suited for answering *always-on* or *continuous* questions. These are questions for example, that ask, what is the rolling average, how many parts have been built so far today, how does production so far today compare with yesterday, and other continuous, sub-second response statistics. Because of the nature of the questions that Streams is designed to answer, it provides join-like capabilities that are well beyond those found in standard SQL. Streams easily and natively handles structured and unstructured data, both in ASCII text and binary.

Note: With a standard relational database, the data is held by persisting it to disk. With IBM InfoSphere Streams, the query (actually a Streams Application) sends the appropriate data directly to the consumer of that data. To run a new query (ask a new question) with Streams, you run a new Streams Application.

To put it another way, all software environments are composed of three major components: process, memory, and disk. With Streams, it is the process component that consumes the data and develops the response to the queries.

2.2 Concepts and terms

At the simplest level, Streams is a software product. It is well integrated with the operating system for high performance, but it is still only software, not a combination of hardware and software. In a world of database *servers*, LDAP *servers*, and J2EE compliant application *servers*, Streams is a runtime and development *platform*. As such, Streams contains:

- ▶ A runtime environment

The runtime environment includes platform services and a scheduler to deploy and monitor Streams Applications across a single or set of integrated Hosts.

- ▶ A programming model

Streams Applications are written in the Streams Processing Language, a largely declarative language where you state what you want, and the runtime

environment accepts the responsibility to determine how best to service the request.

Note: There are many adjectives used to describe computer languages, including the adjective, *declarative*. In a declarative computer language, you describe the results (the computation) you desire, without defining *how* to gather those results (without defining the program flow and control).

SQL, and more specifically SQL SELECT statements, are probably the most widely known example of declarative programming. With an SQL SELECT, you specify only the resultant data set you want to receive. An automatic subsystem to the database server, the query (SQL SELECT) optimizer, determines how to best process the given query, that is, whether to use indexes, or sequential scans, what the table join order should be, the table join algorithm, and so on. The query optimizer decides how to return the resultant query in the fastest time, using the least amount of resource.

In Streams, a Streams Application is written in the Streams Processing Language, which is declarative. A Streams Application defines the (data processing) result that you want. The *how* portion of that execution is determined by the Streams runtime environment (which includes the scheduler), the Streams Processing Language Compiler, and to a lesser extent, a number or property and resource files.

The Streams equivalent to a query optimizer arrives in the form of these three entities.

- Tools and monitoring and administrative interfaces

Streams Applications process data at high speeds, which the normal collection of operating system monitoring utilities cannot handle, for example, processing 12 million messages per second with results in 120 microseconds. So a new processing paradigm is required, and that is Streams.

2.2.1 Streams Instances, Hosts, Host Types, and (Admin) Services

A Streams Instance is a complete, self-contained streams run time. It is composed of a set of interacting Services executing across one or many Hosts.

Figure 2-3 shows a Streams Instance and its member Hosts. As with most software systems, the terms found to exist in a Streams discussion are either physical or logical. *Physical terms* exist, and have mass and are measurable, whereas *logical terms* are typically definitions, and contain one or more physical terms.

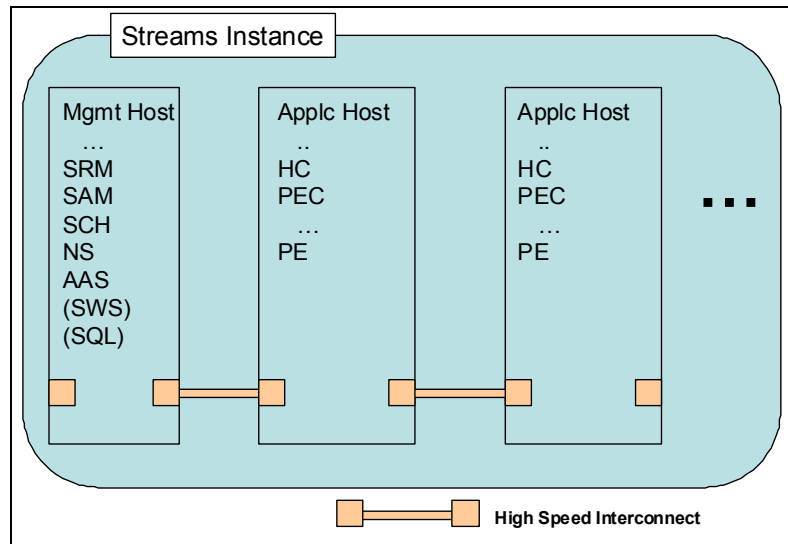


Figure 2-3 Streams Instance, Hosts, and more

The following comments pertain to Figure 2-3:

- ▶ As a term, a Streams Instance is logical, that is, a Streams Instance serves as a container for the physical entities or Hosts.

Streams Instances are started, and then stopped, and the entities contained inside a Streams Instance are then modified, compiled, submitted, deployed, and so on.

From an administrative stand point, a Streams Instance is created and then can be deleted. Creating a Streams Instance is a relatively low cost procedure.

- ▶ As a term, a (Streams) Host is a physical term, and almost nearly and completely equates with a single operating system host. In certain contexts, a (Streams) Host is also called a Node.

In order for it to exist, a Streams Instance (a logical term) must be composed of one or more Hosts (Host being a physical term).

- ▶ A (Streams) Host is exclusively one of three types: a Management Host, an Application Host, or a Mixed-Use Host.

- A Management Host executes the resident services that are the Streams runtime environment proper, such as the Streams Authorization and Authentication Service (AAS) which, among other responsibilities, verifies user identity and permission level.
- An Application Host is dedicated to executing the actual Streams Applications (Processing Elements), which were previously described as being the queries that run continuously inside Streams.
- A Mixed-Use Host executes both resident services (all or a subset of these services) and Streams Applications, and are perhaps more common in development environments.

Note: While a given (Streams) Host could support two or more concurrently operating Streams Instances, that condition is probably not optimal from a *performance* stand point.

There is currently no coordination or communication of any kind between Streams Instances, and as high-end Streams Instances can easily make full use of system resources, locating two or more Streams Instances per Host is likely to produce performance bottlenecks.

- ▶ Management Hosts or Mixed-Use Hosts may provide one or more of the following Streams Services:
 - Streams Resource Manager (SRM)

The SRM Service initializes a given Streams Instance, and aggregates system-wide performance metrics. The SRM Service is also that Service that interacts with the Host Controller of each Application Host.
 - Streams Application Manager (SAM)

The SAM Service receives and processes job submission and cancellation requests, primarily through its interaction with the SCH Service (Scheduler Service).
 - Scheduler (SCH)

The SCH Service gathers runtime performance metrics from the SRM Service, and then feeds information to the SAM Service to determine which Applications Host or Hosts are instructed to run given a Streams Application (Processing Element Containers).

Note: The Streams Scheduler Service determines which Application or Mixed-Use Host runs a given Streams Applications (Processing Element Containers).

The default Scheduler mode is *Balanced*. The Balanced Mode Scheduler distributes a Streams Application (Processing Elements) to the currently least active Host or Hosts.

The Predictive Mode Scheduler uses resource consumption models (CPU and network) to locate Streams Applications (Processing Elements or PEs) on a given Host or Hosts, it being the case that different PEs are known to consume differing amounts of resource.

- The Name Service (NS) provides location reference for all administrative services.
- The Authorization and Administrative Service (AAS) authenticates and authorizes user service requests and access.

- Streams Web Service (SWS)

The SWS Service is the first optional Service listed, and provides web communication protocol access to a given Streams Instance's administrative functions.

- SQL

A Streams Instance has the optional capability to store critical administrative services states inside an SQL based repository for recovery purposes.

- ▶ Application Hosts or Mixed-Use Hosts may provide one or more of the following Streams Services:

- Host Controller (HC)

The HC Service runs on all Application Hosts and Mixed-Use Hosts, and starts and monitors all Processing Elements.

- Processing Element Container (PEC)

The PEC exists as an operating system binary program. Processing Elements exist as custom shared libraries, and are loaded by PECs as required by given Streams Application definitions.

Note: Secure operating systems generally disallow one operating system program from having interprocess communication with another, or from sharing concurrent file or device access. While observed functionality is generally reduced, an upside to this capability is greater process isolation and greater fault tolerance.

To accommodate secure operating systems, IBM InfoSphere Streams supports the concepts of Domains (groups of Application Hosts, or Nodes).

Other terms in this discussion include, Confined Domains, Vetted Domains, and Node Pools, Streams Policy Macros, and Policy Modules. These terms are not expanded upon further here.

- The last items related to Figure 2-3 on page 41 is that it includes the presence of a high speed network interconnect, and any optional shared file system.

This concludes the discussion of the Streams runtime environment. Now we discuss the entities that operate in that environment.

2.2.2 Projects, Applications, Streams, and Operators

In this section, we discuss and describe several of the key terms associated with Stream computing:

- Data is stored in databases, and used in typical database applications, as *records*. In relational terms, the data is stored in rows and columns. The columns are the individual defined elements that comprise the data record (or row). In Streams, these data records are called Tuples and the columns are called Attributes.

The primary reason these items are called Tuples versus records is to reflect the fact that Streams can process both structured (rows and columns) data and unstructured data (variable data representations, flat or hierarchical, and text or binary).

- A Stream is the term used for any continuous flow from a data source. That data is then in the form of Tuples, which are composed of a fixed set of Attributes.
- An Operator is a component of processing functionality that takes one or more streams as input, processes the Tuples and Attributes in the Streams, and produces one or more Streams as output.

- ▶ A Streams Application is a defined collection of Operators, connected together by Streams. A Streams Application defines how the runtime should analyze a set of stream data.

The image shown in Figure 2-4 on page 46 is taken from the IBM InfoSphere Streams Studio, which is the primary developers workbench for Streams:

- ▶ The image is organized into three areas: top, middle, and bottom. The top portion of the image displays a Streams Processing Language source file in the Source Editor View. The middle portion displays the Streams Live Graph View, and the bottom portion displays the Application Graph View.

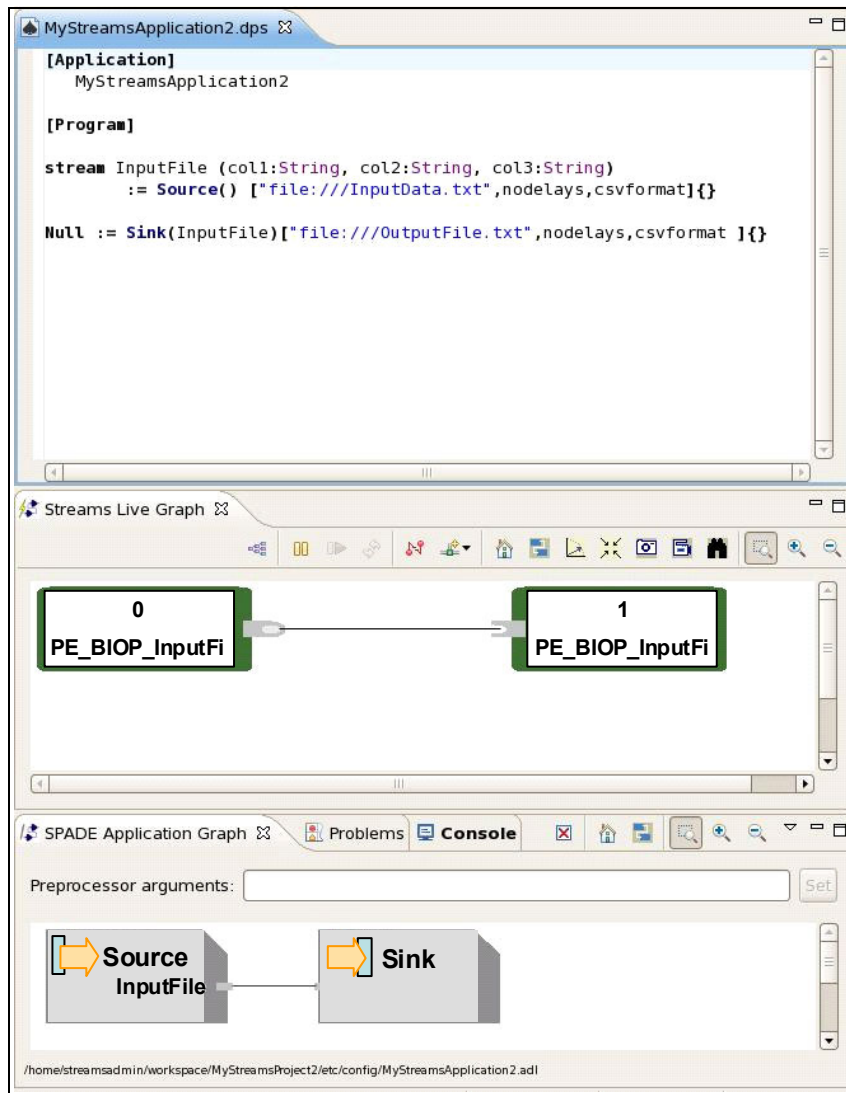


Figure 2-4 Inside Streams Studio

- The Streams Application represents the smallest functional Streams Application one can create. As such:
 - A Streams Application source file is organized into six sections, two that are always present, and four that are optional. When present, these sections must appear in order.

Only the two always present sections are displayed (used), and are identified by the square bracket pairs entitled [Application] and [Program].

- The [Application] section names this given Streams Application, and might optionally contain further modifiers.
- The [Program] section contains the Streams Processing Language declarative commands (Operators) proper, and in Figure 2-4 on page 46, two such Operators are displayed.
- The minimum functional Streams Application defines at least one Source Operator (called a Source) and one destination Operator (called a Sink).

Note: Technically, a Streams Application does not require a destination Operator, and only requires a Source Operator. However, that style of Streams Application might not overly useful.

The Streams Application displayed in Figure 2-4 has its own Source Operator (which refers to an operating system file), and its own Sink Operator (which also refers to an operating system file), and thus provides a *Self Contained Streams Application*.

As more complex Streams Applications gain more and more Operators, both Sources and Sinks, and then also intermediary Operators (not pictured), you may optionally split these larger Streams Applications into two or more Streams Application Source Files. You then can introduce the Stream Processing Language modifiers (modifying a Source Operator) entitled Import and Export to enable these subset Streams Applications to explicitly define their endpoints.

Summarized from above, there are self-contained Streams Applications and there are *Distributed Streams Applications*. Distributed Applications are those that contain Import and Export modifiers.

- ▶ The middle portion of Figure 2-4 on page 46 displays the Streams Live Graph View.

Here you can observe a given Streams Application running live, with row counts progressing from Operator to Operator, along with various colored status indicators, timings, latency measurements, and so on.

- ▶ The bottom portion of Figure 2-4 on page 46 displays the (Streams Processing Language) Application Graph, which is essentially a logical view of a given Streams Application, its Operators, and resultant Streams.
- ▶ A Streams Project (not pictured) serves as a container to a Streams Application (there is a one-to-one correspondence of Project to Application), and contains other properties and configuration settings related to a Streams Application.

- ▶ As a term, the word Stream represents the actual flow of data Tuples from Operator to Operator.
Generally, Source Operators output one Stream, which might be consumed by downstream Operator(s).
- ▶ The definition (count and associated data types) of the data Attributes being consumed by, or output by, a given Operator is referred to as a Schema Definition (Schema).
Generally, most Operators can accept only one input (Schema Definition) and produce only one output (Schema Definition). That is, an Operator can accept multiple input Streams, as long as each matches in its Schema Definition.

Note: While the example Streams Application in Figure 2-4 uses operating system files for its Source Operator and Sink Operator, this is only common for early testing or development of a Streams Application. More common in production is to make use of a variety of TCP/IP (TCP Internet Protocol) or TCP/UDP (TCP User Datagram Protocol) sockets.

Operating system files, and the TCP and other Source and Sink Operators, are called (Streams) *Built-in Adapters*, and are part of the Streams runtime environment.

Streams also includes the Streams Adapters Toolkit, which includes support for ODBC (relational databases), IBM solidDB, WebSphere Front Office (WFO) Source (generally referring to stock market data feeds), and the InetSource Operator (support for HTTP, RSS, FTP, URI, and other communication protocols).

If you still have not found support for your given data source or data target, Streams also allows for user-defined sources and Sinks.

2.2.3 Applications, Jobs, Processing Elements, and Containers

We have previously defined a Streams Application (both self contained and distributed), but that represents only the definition (the declaration) of a given request for service inside the IBM InfoSphere Streams platform. We need further terms to define the instantiation and execution of these objects.

A Streams Application is the definition of a given request for service, submitted for execution in the form of a (Streams) Job. Generally, there is a one to many relationship of applications to Jobs. An application may not be running (zero Jobs), running only once (one Job), or there might be several copies of a given application running (many concurrent applications, or more accurately, Jobs).

Applications can be defined to accept runtime parameters, so that each of the concurrently executing Jobs are performing distinct processing.

Jobs are logical terms, and exist in the form of one or more Processing Elements (PEs). Where the Processing Element Container (PEC) Service runs on each Application Host or Mixed-Use Host, the PEC exists as an operating system binary program. PEs exist as shared libraries that are loaded by PECs and then run.

Note: Ultimately, the discussion of Processing Element Containers (PECs) and Processing Elements (PEs) leads to a discussion of *process architecture*.

Rather than throwing (n) dozen or (n) hundred executable processes at the operating system, and asking the operating system to perform process scheduling (with all of the associated and costly process context switching), the Streams runtime environment performs this critical task.

Ultimately, only Streams can understand how to best schedule its activities, especially considering that Streams has property files and other insights into the contents of its own programming.

2.3 End-to-end example: Streams and the lost child

In this section we detail a given IBM InfoSphere Streams (Streams) Application end-to-end. The business problem used in this end-to-end example is described in this section.

At upscale alpine ski resorts, you might find that your room key not only opens the door to your guest room, but also serves as a charge card at the resort retail outlets, game rooms, and so on, and also allows you access to the ski lifts. Besides tracking customer preferences, this system can also be used to locate lost or missing children.

In the application we model in this section, we receive three input data feeds: door scans, including the guest room, pool, and other areas, retail purchases, which also indicate location, and access to the ski lift lines. The Streams Application we use keeps the most current location of each child on the resort property, and matches that data with requests to find lost or missing children.

In a real situation, these input data feeds would arrive from various live systems. To better enable you to recreate this application in your environment, the example below uses ASCII text files. The sample contents of these files are shown in Example 2-1.

Example 2-1 Sample data used in this example: Input and output

Door Scans		3rd Col here is Boolean 1=Minor			
Lname	Lname		Timestamp	Location	
Morgenson	Sally	5,0,0	2010-02-14 10:14:12	Pool	
Tanner	Luis	12,1,0	2010-02-14 10:14:12	Patio	
Green	Roxanne	1,1,0	2010-02-14 10:14:12	Pool	
Davidson	Bart	0,0,1	2010-02-14 10:14:13	Club House	
Thomas	Wally	0,0,0	2010-02-14 10:14:14	Exercise Room	
Williams	Eric	0,0,0	2010-02-14 10:14:13	Club House	
Ignacio	Juan	3,1,0	2010-02-14 10:14:14	Game Room	
Ryne	Paul	0,0,0	2010-02-14 10:14:13	Club House	
Irwin	Mike	2,2,1	2010-02-14 10:14:14	Laundry Room	

Lift Lines		Y=Minor			
Lname	Fname	Timestamp		Location	
Irwin	Mike	2010-02-14 10:13:55		Pioneer Lift	Y
Davidson	Frederick	2010-02-14 10:13:55		Garfield Lift	N
Samuel	Corky	2010-02-14 10:13:55		Pioneer Lift	N
Ignacio	Juan	2010-02-14 10:13:55		Pioneer Lift	N
Stevens	Sam	2010-02-14 10:13:55		Tumbelina Lift	N
Bartolo	Izzy	2010-02-14 10:13:55		Breezeway Lift	N
Edwards	Chuck	2010-02-14 10:13:55		Pioneer Lift	N
Sylvestor	Hugo	2010-02-14 10:13:55		Tumbelina Lift	Y
Thomas	Wally	2010-02-14 10:13:55		Tumbelina Lift	N

Cash Register Swipes		Y=Minor			
Lname	Fname	Location	Timestamp		
Green	Roxanne	Soda Shop	2010-02-14 09:23:56		
Balmos	Fred	Gift Shop	2010-02-14 10:11:51		
Morgenson	Sally	Cafeteria	2010-02-14 10:14:08		

Child Who is Lost

Lname	Fname	Who-called
Irwin	Mike	Mum
Dewey	Ian	Dad

Results File

Lname	Fname	Who-called	Location	Timestamp
Irwin	Mike	Mum	Laundry Room	2010-02-14 10:14:14
Dewey	Ian	Dad	-	-

Where:

- ▶ Five total files are displayed.
- ▶ Three input files, Door Scans, Lift Lines and Cash Register Swipes, represent normal guest activity at the alpine ski resort, such as passing through a secure entrance way, purchasing something, or entering the lift line.

Each of these three files is modelled differently, with different Attribute order and data types. A multi-Attribute primary key is present in the form of last name (lname) and first name (fname) combined.

A mixture of one/zero, and Y/N, represents the condition of children versus adults. This system is designed only to locate missing children.
- ▶ A fourth input file (Child Who is Lost) contains requests by parents to locate their missing child.
- ▶ The fifth file represents output, a result set.

In the case of Mike Irwin, the system reported a last known location, whereas the system did not have a known location for Ian Dewey.

2.3.1 One application or many

In the sample Streams Application shown in Example 2-2 on page 52, we create this solution as one Application (a self-contained Streams Application). In a real situation, it would be more common to organize this (system) into several Streams Applications (a distributed Streams Application).

Consider the following:

- ▶ A Distributed Streams Application will have minimally an Application that is upstream, outputting Tuples, and one or more Applications that are downstream, consuming those Tuples.

The Streams modifiers (language keywords) that create this capability syntactically are Export and Import.

- ▶ The association between Exporting Application and Importing Application is either done *point-to-point* (explicit, named reference) or by *publish and subscribe* (implicit, by named topic and category).

In either case, the physical transport mechanism (physical communication method) is unchanged.

- ▶ In the example above, each input topic would likely have its own Streams Application.

If retail sales (cash registers) are only active 18 hours a day, why then run the communication system to receive that data all day long? Door scans, however, do run and do report data all day long.

Example 2-2 Self-contained Streams Application

[Application]

MyApplication_S

[Program]

```
vstream doorScanSchemaIn
(
  lastName  : String,
  firstName : String,
  unused1   : Short,
  unused2   : Short,
  ifIsMinor : Short,
  timeStamp : String,
  location  : String
)
```

```
vstream liftLineSchemaIn
(
  lastName  : String,
  firstName : String,
  timeStamp : String,
  location  : String,
  ifIsMinor : String
)
```

```

vstream registerSwipeSchemaIn
(
  lastName  : String,
  firstName : String,
  location  : String,
  timeStamp : String,
  ifIsMinor : String
)

vstream mergeSchema
(
  lastName : String,
  firstName : String,
  timeStamp : String,
  location  : String,
  ifIsMinor : String
)

vstream outputSchema
(
  lastName : String,
  firstName : String,
  timeStamp : String,
  location  : String
)

# #####

stream inputDoorScan ( schemaFor (
  doorScanSchemaIn ) )
:= Source ( )
[
  "file:///DoorScans.txt",
  nodelays,
  csvformat
]
{ }

stream mergeDoorScan ( schemaFor (
  mergeSchema ) )
:= Functor ( inputDoorScan )
<
String $ifIsMinorFlag := "_" ;
>
<

```

```

        if (ifIsMinor = 1s) {
            $ifIsMinorFlag := "Y" ;
        } else {
            $ifIsMinorFlag := "N" ;
        }
    }
    >
    [ ifIsMinor = 1s ]
    {
        # lastName := lastName,
        # firstName := firstName,
        # timeStamp := timeStamp,
        # location := location,
        ifIsMinor := $ifIsMinorFlag
    }

stream inputLiftLine ( schemaFor (
    liftLineSchemaIn ) )
:= Source ( )
[
    "file:///LiftLines.txt",
    nodelays,
    csvformat
]
{ }

stream mergeLiftLine ( schemaFor (
    mergeSchema ) )
:= Functor ( inputLiftLine )
[ ]
{ }

stream inputRegisterSwipe ( schemaFor (
    registerSwipeSchemaIn ) )
:= Source ( )
[
    "file:///RegisterSwipes.txt",
    nodelays,
    csvformat
]
{ }

stream mergeRegisterSwipe ( schemaFor (
    mergeSchema ) )
:= Functor ( inputRegisterSwipe )
[ ]

```



```

{ }

# #####

stream stream_S1 ( schemaFor ( outputSchema ) )
  := Functor ( mergeDoorScan, mergeLiftLine,
               mergeRegisterSwipe )
  [ ifIsMinor = "Y" ]
  { }

stream stream_S2 ( lastName_c : String,
                  firstName_c : String, caller : String )
  := Source ( )
  [
    "file:///LostChild.txt",
    nodelays,
    csvformat,
    initDelay=5
  ]
  { }

# #####

stream joinedRecords ( lastName : String ,
                      firstName : String, caller : String,
                      location : String, timeStamp : String )
  := Join( stream_S2 < count ( 0 ) >;
          stream_S1 < count ( 1 ), perGroup > )
  [
    LeftOuterJoin, { lastName_c } = { lastName },
    firstName_c = firstName
  ]
  {
    lastName_c,
    firstName_c,
    caller,
    location,
    timeStamp
  }
Null := Sink ( joinedRecords )
[ "file:///OutputFile_S.txt", csvformat, nodelays ] { }

```

2.3.2 Example Streams Processing Language code review

The following is a code review related to Example 2-2 on page 52:

- This Streams Application source code file has the two always present sections of [Application] and [Program].

The [Application] section names this Streams Application, which is named `MyApplication_S`.

The [Program] section contains Streams Processing Language declarative command proper, and comprises the majority of this file.

- There are five (initial paragraphs, delimited by white space) that start in this [Program] section. Each begins with the keyword *vstream*.

Note: A *vstream* is a Virtual Schema Definition, or (referenceable and reusable) Schema Definition. Generally, a Virtual Schema Definition is equal in all aspects to an explicit Schema Definition.

An example of an explicit Schema Definition is available in Example 2-2, on the Streams Processing Language source code line that begins with `stream joinedRecords`.

Each of these five paragraphs define a Schema Definition, which is essentially a Tuple definition. Any of these Schema Definitions could have been placed inside a given Operator and been *functionally* equivalent. The purpose of defining these once at the start of the Application is to increase readability and code reuse, as some of the Schema Definitions are used multiple times inside this file.

- A Schema Definition Attribute mapping has the general form or Attribute name, Attribute data type, (comma and repeat). The Attributes data types presented in Example 2-2 on page 52 include String and Short.

Note: Streams Processing Language includes 10 or more simple data types, and also an associated *list data types* (basically arrays of simple data types).

Reading data: Source Stream Operator

The first real line of Streams Processing Language source code in Example 2-2 on page 52 begins with the words `stream inputDoorScan`, and ends with the pair of trailing curly braces “`{ }`”. Regarding this code:

- ▶ This line defines one Source (input) Stream (an Operator) to this Streams Application. This Operator happens to be a ASCII text file reader, based on its syntax.
- ▶ This line begins as `stream`, stream name (with a stream name we chose and named `inputDoorScan`).

If any further Operator in this Application wants to consume the output of this Operator, it could do so by using the Stream Name `inputDoorScan`. Generally, any Operator can automatically support multiple consumers.

- ▶ The Stream name is followed by the output Attributes specification to this Operator. We could explicitly specify Attribute names and Attribute data types, but optionally make reference to the Virtual Schema Definition defined above (`doorScanSchema`).

- ▶ The square bracket pair “`[]`” of this source code line specify modifiers to this Operator; specifically, we set the file name, and the fact that this input file is variable spaced (as opposed to fixed space) and delimited with commas.

By default, each inputted data Tuple would receive a time stamp, which we do not need for this application. The modifier named `nodelays` suppresses that default IBM InfoSphere Streams runtime environment behavior.

- ▶ The trailing curly brace pair “`{ }`” can optionally contain the output Attribute mapping, as output Attribute values can be derived. Because this Operator receives (n) Attributes that we do not alter, we can accept the default mapping and leave this curly brace pair empty.

Filtering and mapping Attributes: Functor Operator

The second real line of Streams Processing Language source code in Example 2-2 on page 52 begins with the words `stream mergeDoorScan`, and ends with the pair of trailing curly braces “`{ }`”, although these curly braces contain values. Regarding this code:

- ▶ This Streams Operator is of type Functor. In its simplest form, a Streams Processing Language Functor Operator is similar to a transform Operator in a standard extract-transform-load software package.
- ▶ This Streams Operator receives five input Attributes, as determined by its Source Stream (`inputDoorScan`);. The Source Operator (and Source Stream) sends five Attributes, so we receive five Attributes. These Attributes are referenced by their Attribute names and Attribute data types, as specified in the Source stream.

Note: This Stream Operator also outputs five Attributes, as determined by its Schema Definition, which is named mergeSchema.

These Source and output Schema Definitions for this Operator are equal in Attribute count, and also the collection of Attribute names. These two Schema Definitions do differ, however, in the Attribute data type for the last Attribute, which is named ifIsMinor. We change (recast) that data type from Short to String to demonstrate this technique (and also to make the definition of this data Stream equal the other two data Streams we manipulate in a related manner in the following sections).

The trailing pair of curly braces lists only one output Attribute expression, yet five total output Attributes are expected. The first four output Attribute expressions are listed, but commented out, and are thus not truly present. The Streams Processing Language Compiler will map these four missing Attribute expression for us by Attribute name.

- ▶ In Example 2-2 on page 52, there are two sections (blocks) of Streams Processing Language statements delimited by angle bracket pairs, “< >”. Regarding this code:
 - The first of these sections defines and initializes a Functor Variable named ifIsMinorFlag.
 - The second section compares the value of the input Attribute named ifIsMinor to the literal value of “1”. A letter “s” immediately follows (no white space) the literal value 1, to cast this value as a type Short.

Note: In Streams Processing Language, all Attributes and all Attribute assignments and comparisons are explicitly cast.

If the value of this input Attribute equals 1, we set the Functor Variable named ifIsMinorFlag, to equal “Y”; otherwise, it is set to “N”.

Note: Algorithmically what we are doing here is to recast this input Attribute data type.

Because computers cannot assume that a numeric one should translate to the upper case letter “Y” (Yes), and numeric zero translate to “N” (No), we have this block of statements.

- These two blocks of angle bracket delimited Streams Processing Language statements are adjacent, but must be entered as two distinct blocks, not one (merged) block.

The first block contains a variable definition, whereas the second contains a variable expression, and thus must be entered and maintained separately.

- Lastly, the square bracket pair with the expression “iflsMinor = 1s” acts as a filter to this Functor Operator. Input Tuples, and more explicitly, the input Attribute entitled iflsMinor, is evaluated as containing the value 1 or (non 1). Value equal to 1 Tuples are permitted through this Functor Operator, whereas (non 1) Tuples are not.

Note: We run this same evaluation expression later for our two other main input Stream Operators (for stream Stream_S1), and per our source code specification, perform this evaluation redundantly for this same Stream (of merged) data.

We perform this task merely to detail the numerous means to perform this technique.

Four more Operators

The next four Streams Operators in this Streams Application (stream inputLiftLine, stream mergeLiftLine, stream inputRegisterSwipe, stream mergeRegisterSwipe) are of equal or lesser functionality than the most recent two Streams Operators we detailed above.

These four Streams Operators define, and then Attribute map (remap or change the Attribute order) of two new input Streams and are required for our fuller example, but demonstrate no new techniques. The next significant Operator we detail is stream_S1.

Merging (input) Streams: (Any) Operator

The Streams Processing Language statement that begins with the words `stream stream_S1` is also an Operator of Type Functor. We need an Operator of type Functor to apply a filter condition (iflsMinor = “Y”). While our first input Stream was filtered, our two new input Streams are not. This Functor Operator gives us our three main, merged input data Streams.

Note: In a real situation, this Streams Application would start, and run, for a long time. In this simple example, we are using operating system text files with only a few lines of data. The challenge in testing or developing in this manner is the sequencing of multiple operating system programs and multiple Streams PECs.

In this example, we are reading from four small input files, and it could happen that we read from the single file containing the request to find a lost or missing child (and perform the join) before we ever load the three files that contain the current child location (and thus return zero joined Tuples).

To better accommodate development and testing on such a small sample, the Streams Processing Language source code line that begins `stream_S2` displays the modifier `initDelay = 5`. This modifier causes this Source Stream to wait 5 seconds before completing its initialization.

Before proceeding to the next significant Operator in this Streams Application, we must also open the input data Stream that received the request to locate lost or missing children. This Operator is `stream_S2`.

Note: At this point in our Streams Application, we have three data sources filtered, and Attributes mapped (re-mapped), into one continuing data Stream. These data Attributes are our Tuple of which child was last seen on the alpine ski resort, and also when and where.

The next challenge (business requirement) in the Streams Application is to Join this data with any requests to find lost or missing children.

Joining, aggregating, or sorting static (normal) data

The next requirement in our Streams Application is to Join the Tuple of the most recent child locations with an actual request to locate a child. This task is completed with a Streams Processing Language Join Operator. However, joining, aggregating, or sorting streamed data is fundamentally different than joining or aggregating data in a static data repository. In this section, we first detail that challenge.

When sorting a standard list of data, you *cannot* output any of the Tuples until you have first received *all* of those Tuples.

Given the statement above, what if you are receiving a list of Tuples to be sorted alphabetically, but the last Tuple to be received starts with the letter “A” (and should be output first)? Sorting data has this requirement, and so does aggregating data, which has an implied sort; aggregating by Department, you first sort the list of Company-wide data by Department.

Joining data also implies this condition, because you nearly always sort at least one side of the joined lists to achieve better performance.

The assumption above, however, is that you know where or when the end of the data is located/received. Remember, you need to receive the last row before you can complete a sort. Because streaming data never terminates, how then do you sort (or aggregate, or join)? The answer is to *create windows*, or *marked subgroups* of the data received.

With IBM InfoSphere Streams, an input data Stream might never terminate. The inputted data might arrive and already be marked into subgroups, or it might not. And yet, you still need to join data, aggregate data for given (subgroups), or sort data within given subgroups. To accomplish this program requirement, Streams defines the concept of (moving) *windows* of data.

Joining, aggregating, or sorting streamed data

In an endless Stream of data, IBM InfoSphere Streams allows data to be promoted into subgroups called windows (of data) by a number of conditions. The following is a list of some of those conditions:

- ▶ By row count.
- ▶ By elapsed time.
- ▶ By a pre-existing marker in the data (this marker is referred to as a “Punctor”, or punctuation). An upstream Operator in Streams can also add punctuation.
- ▶ By an evaluation expression in the data proper.
- ▶ A combination of the above conditions.

Note: Other software environments call these subgroups of data Waves. They also have stand-alone Operators to generate this condition, which are called Wave Generators.

Inside IBM InfoSphere Streams, windows and windows definitions are elements of a given Streams Operator, and are not stand-alone Operators themselves.

With Streams, there are two types of windows, Tumbling and Sliding. A *Tumbling window* fills with data, performs or supports whatever Operator it is an element

of, and then moves to an entirely new window (an entirely new sub grouping) of data. It *tumbles* forward through the data, end over end. A *Sliding window* fills with its first subgroup of data, performs or supports whatever Operator it is an element of, and then continues with this same data, adding to or replacing it incrementally, or sliding, as new data is received.

For example, a Tumbling window could be used to aggregate streaming data, and report every time a new key value pair has changed its descriptive Attributes. From our alpine ski resort and lost or missing child example, the windows would report every time a child's location changes.

This window is tumbling because it dumps one key value pair Tuple in favor of another. For example, consider the following code:

```
stream tumbling_2 (lname : String , fname : String , location String )
:= Aggregate ( tumbling_1 < count ( 1 ) , perGroup > ) [ lname . fname ]
{ Any ( lname ) , Any ( fname ) , Last ( location ) }
```

With regard to the previous single Streams Processing Language statement, consider the following items:

- The single Streams Operator is of type Aggregate, and includes a required and embedded window specification.

The window type used is Tumbling. Tumbling windows by count generally have only one *count* keyword (one *time* keyword, for example) in their specification. Sliding windows generally have two count specifiers, one for the initial window size and one for the slide size.

- The *perGroup* keyword says that there will be one window for each primary key value of the received rows. The primary key Attribute or Attributes are specified inside a square bracket pair. In this example, we have a two Attribute primary key of last name (lname) concatenated with first name (fname).

If this Operator only ever receives 10 unique primary key values, then there will only ever be 10 concurrent windows.

- The curly brace pair details the output Attribute expressions. If, in this case, the two primary key Attributes do not change, the value for location will be the last (Last) one received. Per the source code specification, this Aggregate Operator will report the current location.

Note: As previously stated, Streams windows are used with at least three Streams Operators, including Aggregate, Sort, and Join. You might choose to model a window using Aggregate or Sort, and then apply this definition to the Join Operator, which simultaneously manages two Streams, which are the two sides of the join pair.

Much like standard relational databases, Streams supports full joins, and left and right outer joins. Streams also allows joins on more than just relational algebra meaning that a left side expression can evaluate to true, and it will join with all right side expressions that are also true, without either side having to be equal to each other.

Further, Streams supports full intersections of data. This means that, by default, Streams attempts to join all Tuples from the left side of the join pair to the right side of the join pair, and *then* joins all right side Tuples with the left side Tuples.

In our example use case, we want an outer join that returns the child's location if it is known, or returns null. We still want a response, even if it is null. However, we only want to join a child identifier with location. In Streams terminology, this is referred to as a *one-sided join*.

Joining Tuples: Join Operator

At this point, we have met all of the prerequisites for our Streams Join Operator, and continue with Example 2-2 on page 52. This discussion continues with the Source line that begins with `stream joinedRecords`:

- ▶ This Operator uses an explicit schema definition. We could have used our standard Virtual Schema and referenced it, but we want to use explicit schema definitions.
- ▶ The windows expression proper is in the parenthesis following the Join keyword. It states the following:
 - `stream_S2` is the dominant Stream (table), meaning it is the one that initiates the Join operation. As an outer Join, it will always report Tuples. This Stream is dominant because it is listed first. If `stream_S2` were listed second, it would be the subservient Stream.
 - `stream_S2` lists a windows size of zero Tuples. That condition makes this a one sided Streams Join.
 - `stream_S1` is listed with a windows size of 1 Tuple and *perGroup*.

The *perGroup* modifier means there will be a series of windows, one per unique key value pair. The single or set of key Attributes are specified by

name, on the right side of the equal signs, in the curly brace pair that follows.

This is a Sliding window specification even though there is only one count keyword for either side of the join pair. *All Join Operators use only Sliding windows*, as it the nature of a join. Join Operators with two-sided Tumbling windows would rarely have elements to join.

Each new key value pair slides the window. In the case of this one-sided join, each new dominant Stream Tuple causes a full scan outer join into the subservient Stream.

Note: Because the subservient Stream keeps the most recent location per child, and the arrival of a new location request triggers the join, we have accomplished our design intent.

Writing data: Sink Operator

The last Streams Operator in Example 2-2 on page 52 (“Null := Sink”) writes the observed result to an operating system ASCII text file, with sample results displayed at the bottom of Example 2-1 on page 50.

While we write to file here to simplify this example, in a real situation, we could have several concurrent subscribers to this (output) Stream, including electronic pagers, event alerters, text message generators, and so on.

Note: The “ := ” Operator expression means that we are performing a Attribute mapping (the colon symbol) and also performing value assignments (the equal symbol).

2.4 IBM InfoSphere Streams tools

IBM InfoSphere Streams offers a combination of command-line tools and graphical tools (web-based and fat client), as well as tools to perform administrative and monitoring functions, and application development functions, as examples. In this section, we detail two of those tools:

- ▶ IBM InfoSphere Streams Studio

An Eclipse-based developer’s and administrator’s workbench.

- ▶ streamtool

A command line interface program to perform many tasks, such as listing Streams Instances, and starting and stopping Instances.

2.4.1 Creating an example application inside Studio

In this section, we provide the steps used to develop the sample Streams Applications displayed in Figure 2-4 on page 46 and Example 2-2 on page 52 using the Streams Studio developer's workbench. This should give you a better understanding of IBM InfoSphere Streams Studio. In the development of the sample Application, the following assumptions are made:

- ▶ The complete IBM InfoSphere Streams product is installed, including a base Eclipse install with Streams Studio plug-ins.
- ▶ The operating system platform is Red Hat Enterprise Linux® version 5.3.
- ▶ For Example 2-2 on page 52, you have created the four sample input files displayed in Example 2-1 on page 50.

Perform the following steps:

1. IBM InfoSphere Streams Studio exists as a collection of plug-ins to the Eclipse developers workbench. To run Studio, launch Eclipse.

Note: Eclipse is an open source and extensible developer's workbench. As IBM InfoSphere Studio is based on Eclipse, it inherits many Eclipse terms and ideas.

In Eclipse, a given parent directory contains all of your Projects, and any metadata related to work bench preferences. In Eclipse, this parent directory is called your *Workspace*. Project is also an Eclipse term, and is thus inherited by Streams. In Streams, there is a one to one relationship of Streams Project to Streams Application.

If you are prompted to select a Workspace when launching Eclipse, select the default value and click **OK**.

If this is the first time you are running Eclipse, you might receive a Welcome window. Feel free to close that view.

2. You need to make a small number of Streams objects before you can begin entering Streams Processing Language statements. Perform the following steps:
 - a. From the Studio menu bar, select **File** → **New** → **Other**.
 - b. Select **InfoSphere Streams Studio** → **Spade Application Project** → **Next**.

Note: Streams Processing Language was formerly known as Streams Processing Application Declarative Engine (SPADE).

- c. Give this new Streams Project a name and click **Finish**.

You might be prompted to change the Perspective; if so, click **Yes**.

Note: There are two additional Eclipse terms that we use: Perspective and View.

A View in Eclipse is a distinct area of the screen, and in other environments might be called a panel or frame. There are so many Views in Eclipse that they are organized into groups of Views called Perspectives.

Streams Studio provides two Perspectives with its Eclipse plug-ins, and a number of Views.

If this is the first time you are running Eclipse and creating a Streams Project, you might receive a prompt to change Perspectives. If so, click **Yes**.

3. At this point, you are ready to enter Streams Processing Language statements.

To begin, you may first create either the sample Streams Application shown in Figure 2-4 on page 46 or Example 2-2 on page 52. To do so, enter the displayed Streams Processing Language statements in the Editor View. As you do, be aware of the following items:

- If you produce an error in your syntax, a red marker will highlight the source of the problem.
For a more detailed description of the error, you may also move to the Problems View.
- While entering statements in the Editor View, you can enter Ctrl-Spacebar to produce context sensitive help, that is, you are prompted with a next option or continuation of the current command or modifier.
- You are done when your Streams Processing Language Application matches the statements you have been entering from the example and contains zero syntax errors.

4. To run the Streams Application, perform the following steps:

- a. In the Editor View, right-click and select **Run as** → **Start Streams Instance**.

On iterative executions of this Streams Application, you do not need to complete this step.

- b. From the Editor View, right-click and select **Run as** → **Submit (...)**.

This step will execute your Streams Application. As Streams Applications never terminate, you should select **Run as** → **Cancel** between each new Application execution.

In the following pages, we show a number of images from the IBM InfoSphere Streams Studio. For example, Figure 2-5 shows the Project Explorer View.

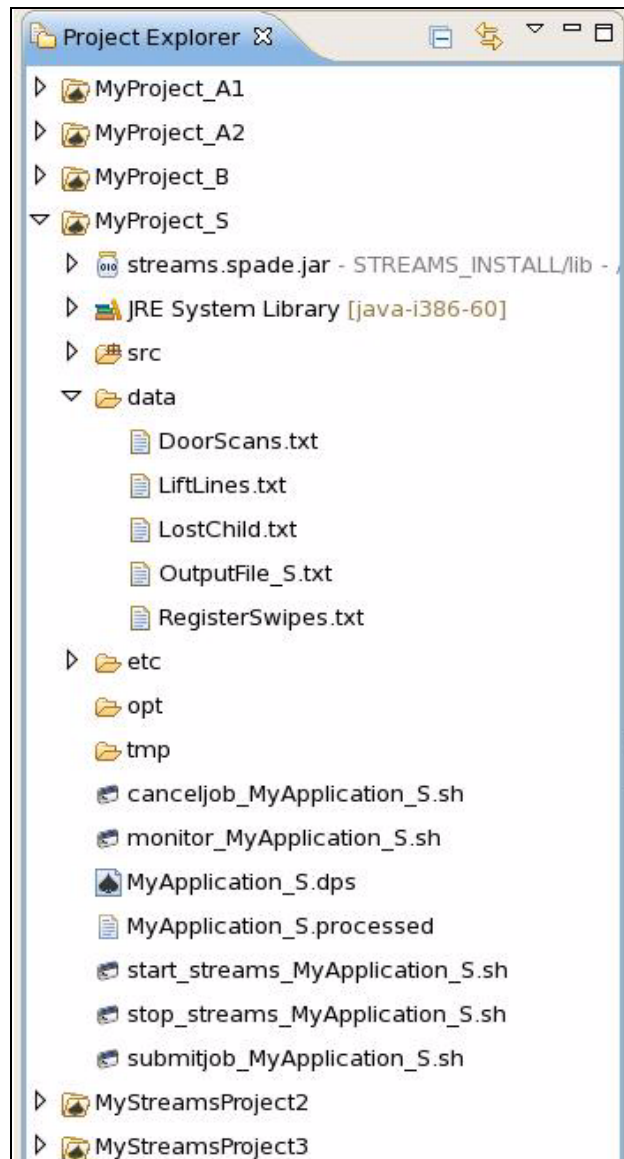


Figure 2-5 Project Explorer View from IBM InfoSphere Streams Studio

The following comments refer to Figure 2-5 on page 68:

- ▶ The Project Explorer View operates like most file explorer interfaces, that is, the highest level object displayed in this View are Streams Projects, followed by the given contents of the Project, the Stream Processing Language source file, and so on.
- ▶ Figure 2-5 on page 68 shows a data subdirectory where all Source and target operating system data files are located.
- ▶ Figure 2-5 on page 68 also shows the Streams Processing Language Application, which is always in a file that has a “.dps” suffix.
- ▶ Other files displayed in Figure 2-5 on page 68 contain configuration settings, and commands to start and stop the given Streams Instance and to submit and cancel the associated Streams Application (Job).

Figure 2-6 shows the Streams Application Graph View from IBM InfoSphere Streams Studio.

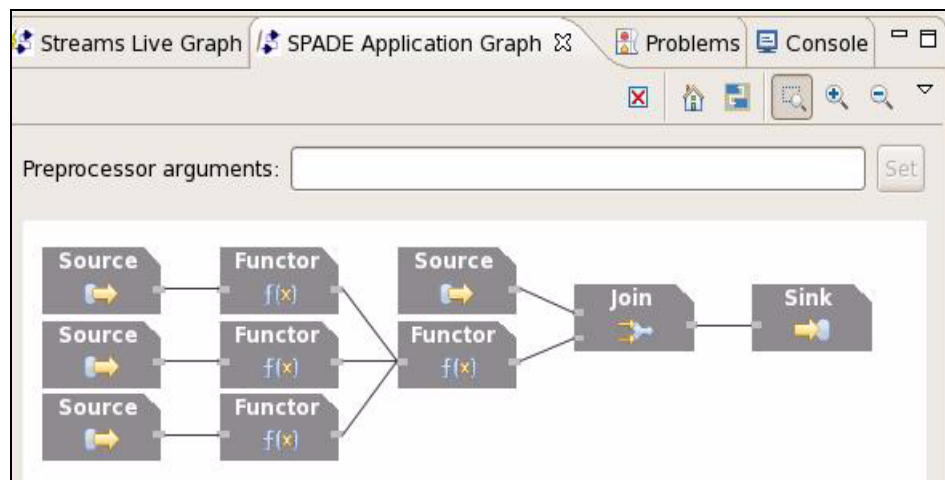


Figure 2-6 Streams Application Graph View from IBM InfoSphere Streams Studio

The following comments refer to Figure 2-6:

- ▶ The window shown in Figure 2-6 shows the logical design of the given Streams Application from an Operator and Stream perspective. It is called a live graph, and is from the sample Application in Example 2-2 on page 52.
As you highlight any of the given Operators, your cursor will move to that point in the Streams Application source file.

- You can use the Streams Application Graph to ensure that all of your Operators are connected as you intend them to be. An Operator with no input/output Stream most likely indicates you have misspelled a given Stream in your Streams Processing Language Application file.

Where Figure 2-6 on page 69 shows the logical design of a given Streams Application, Figure 2-7 shows it executing.

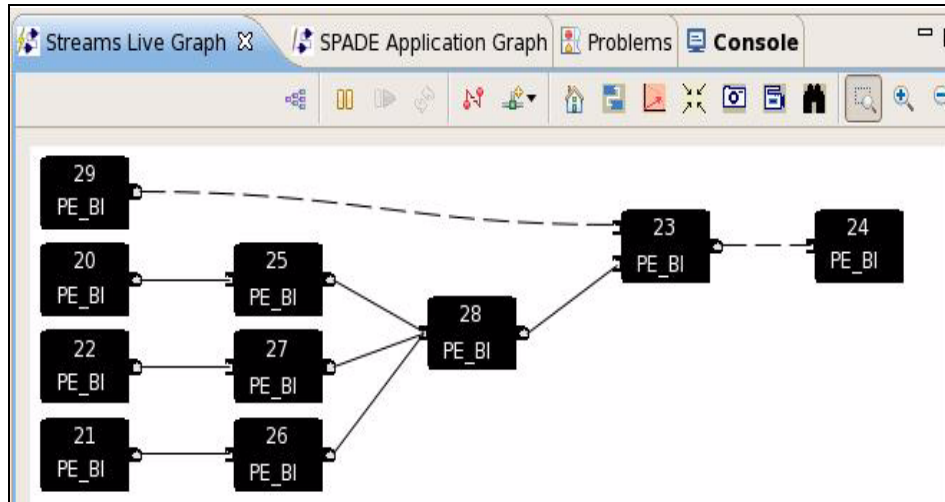
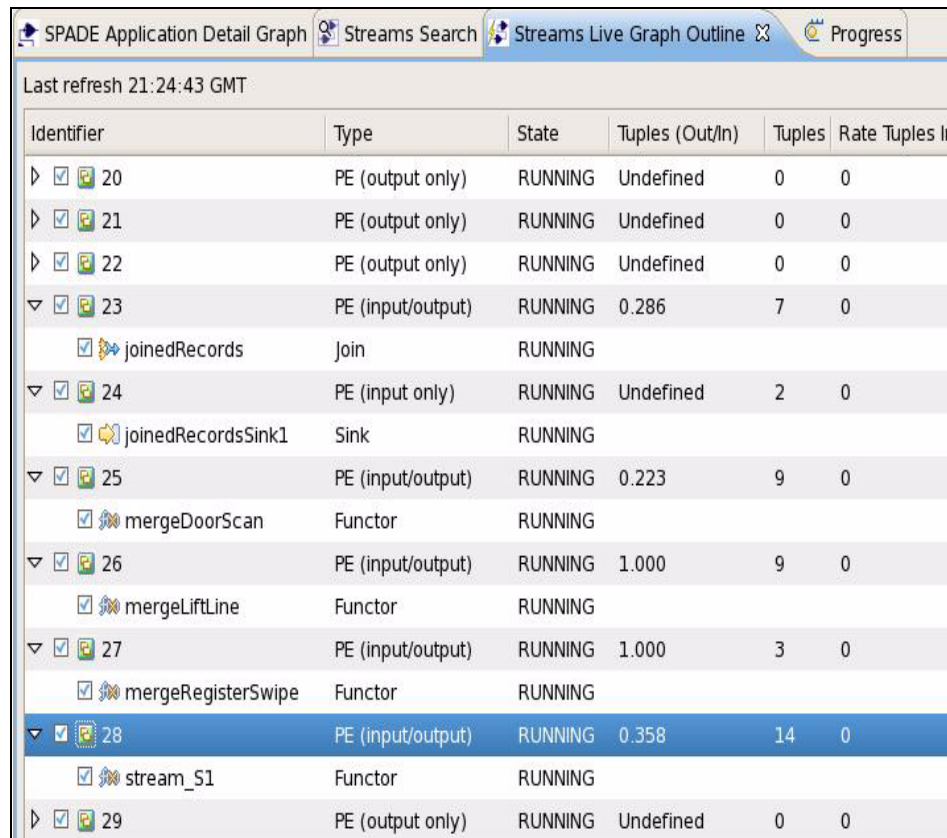


Figure 2-7 (Streams) Live Graph View from IBM InfoSphere Streams Studio

The following comments refer to Figure 2-7:

- The colors of the displayed Operators change as the Operator state changes. If you are not seeing the color changes, this is most likely due to the fact that this Application runs quickly.
- The dashed Stream line from Operator 29 to 23 indicates that this Stream is waiting. This is the Stream of the requests to locate lost or missing children, and is currently in its `initDelay 5` state (a 5 second wait).

The Streams Live Graph Outline View is shown in Figure 2-8. It shows the most detailed, low-level, performance statistics.



Identifier	Type	State	Tuples (Out/In)	Tuples	Rate	Tuples In
▶ 20	PE (output only)	RUNNING	Undefined	0	0	
▶ 21	PE (output only)	RUNNING	Undefined	0	0	
▶ 22	PE (output only)	RUNNING	Undefined	0	0	
▼ 23	PE (input/output)	RUNNING	0.286	7	0	
joinedRecords	Join	RUNNING				
▼ 24	PE (input only)	RUNNING	Undefined	2	0	
joinedRecordsSink1	Sink	RUNNING				
▼ 25	PE (input/output)	RUNNING	0.223	9	0	
mergeDoorScan	Functor	RUNNING				
▼ 26	PE (input/output)	RUNNING	1.000	9	0	
mergeLiftLine	Functor	RUNNING				
▼ 27	PE (input/output)	RUNNING	1.000	3	0	
mergeRegisterSwipe	Functor	RUNNING				
▼ 28	PE (input/output)	RUNNING	0.358	14	0	
stream_S1	Functor	RUNNING				
▶ 29	PE (output only)	RUNNING	Undefined	0	0	

Figure 2-8 (Streams) Live Graph Outline View from IBM InfoSphere Streams Studio

The following comments refer to Figure 2-8:

- ▶ The Streams Live Graph Outline View displays rows counts, bytes, and so on, Operator by Operator. This View is handy for debugging and for monitoring your Streams Application.
- ▶ You can also concurrently run this View with the Streams Live Graph, moving back and forth as you click various elements.

2.4.2 Using streamtool

The IBM InfoSphere Streams streamtool offers a command-line interface that can also be used to accomplish all the steps we performed in previous sections. Most invocations in the streamtool require that you supply a Streams Instance id-name. If you do not know your Streams Instance id-name, you can run the following command:

```
streamtool lsinstance
```

For clarity, lsinstance is “L S Instance” in lowercase and is one word.

Note: If your Streams Instance id-name contains a special character, for example, @, then you must always enter this id-name inside a pair of double quotes. Otherwise your operating system command interpreter will observe and process those special characters.

The above command reports all currently running Stream Instances on the given server. If you do not know your Stream Instance id-name and that Instance is not currently running, you may then start the Instance inside IBM InfoSphere Streams Studio, and then return to this point.

To start or stop a given Streams Instance, run the following command:

```
streamtool startinstance -i id-name
streamtool stopinstance -i id-name
```

Note: It is common for new developers to exit the Stream Studio design tool and leave their Stream Instance running, and then not know how to terminate or even restart the Streams Instance.

To force a shutdown of the Streams Instance, use the **streamtool stopInstance** command and append the “-f” argument (minus f).

By default, the Streams Instance SWS service is not running, but is still required to run the IBM InfoSphere Streams Console, which is a web-based application. To start the Streams SWS service, enter the following command:

```
# Get list and location of all running Streams Services
#
streamtool lshost -i id-name -long
#
# Sample output below, indicating that SWS is not running
#
Host          Services
host-x        hc, aas, sam, sch, srm
```

```
#
# To start the SWS Service enter,
#
streamtool addservice -i id-name --host host-x sws
```

To get the fully qualified URL allowing access to Stream Console, enter the following command:

```
streamtool geturl -i id-name
```

Then enter the URL inside a supported web browser, as shown in Figure 2-9.

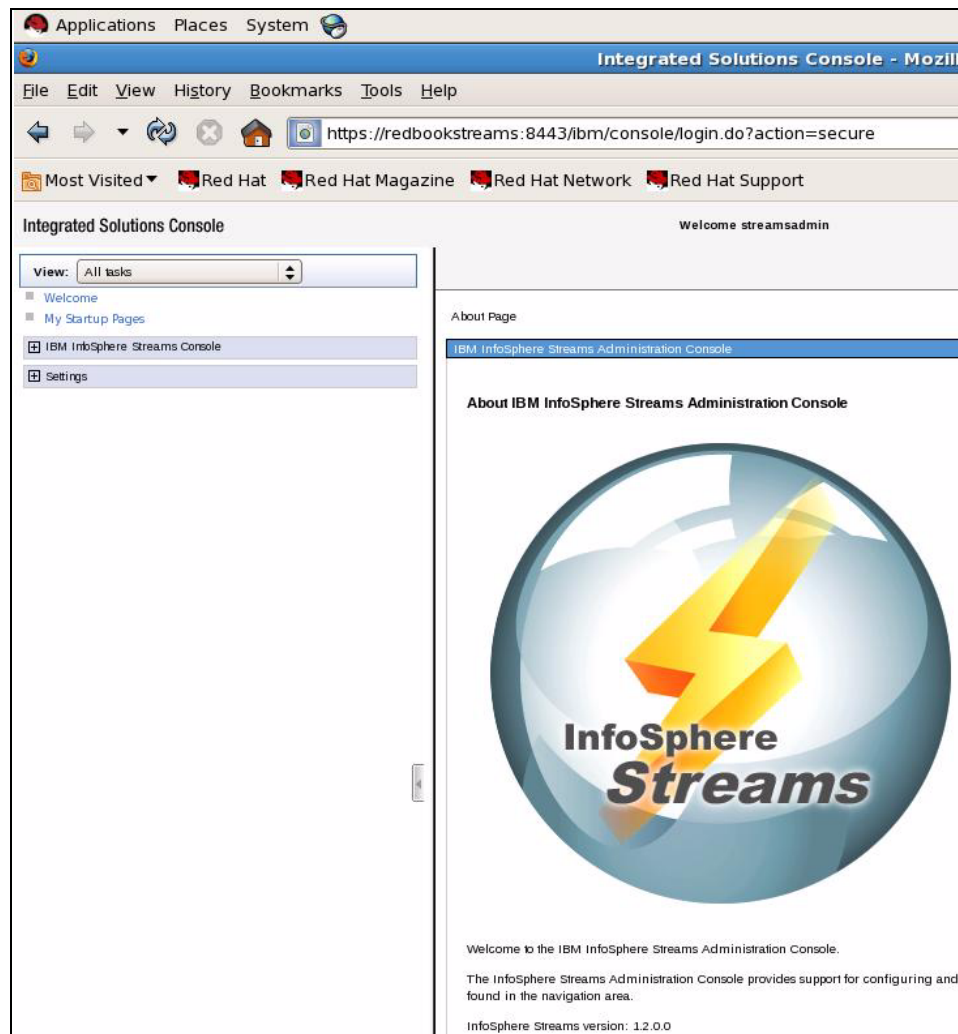


Figure 2-9 IBM InfoSphere Streams Console



IBM InfoSphere Streams Applications

In previous chapters of this book, we introduced IBM InfoSphere Streams (Streams) Applications, which are composed of a combination of data streams and Operators for these streams, which allows Streams to ingest, process, and output data.

In this chapter, we provide a closer look at Streams Applications, and outline the activities required to design a well-performing application. The design process will lead you to consider the purpose, inputs, outputs, and dependencies of the system in question. We also outline features to address the performance requirements that should be considered during application design.

In addition, we also contrast Streams Application development with traditional application development and highlight a number of novel Attributes of Streams Applications.

From early experience with implementing Streams Applications, we list a number of applications classes and design patterns that should help you to understand the Streams platform and allow new developers to better relate their requirements for established systems.

3.1 Streams Application design

Let us assume that you have been tasked with producing a business requirement using the IBM InfoSphere Streams platform. The first thing you need to do is develop a high level system design that identifies the main components of the system.

As shown in Figure 3-1, there are two main components to this design:

- ▶ A logical design identifying the application components to be produced. The logical application design is discussed in the subsequent sections of this chapter.
- ▶ A physical component design identifying the runtime components and hardware platform required to support your application and deliver the level of performance that is required. This physical component design is discussed in Chapter 4, “Deploying IBM InfoSphere Streams Applications” on page 127.

At run time, the Streams Scheduler determines how to efficiently segment the application across the hardware hosts. The application should be segmented to produce suitably sized deployable Processing Elements for the Streams Scheduler to deploy. In 4.4, “Deployment implications for application design” on page 137, we discuss the aspects of appropriate application segmentation.

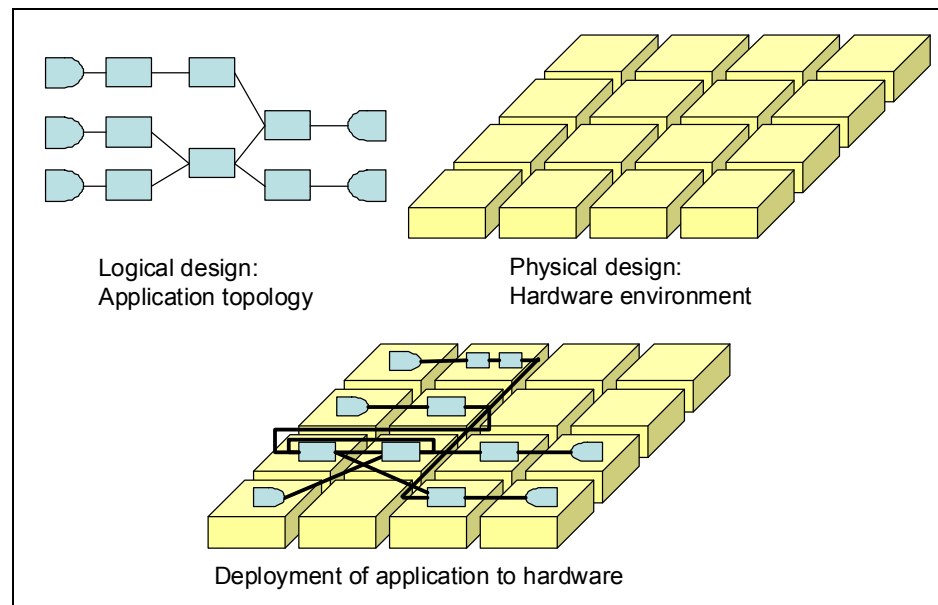


Figure 3-1 Application design, physical design, and deployment

3.1.1 Design aspects

We begin the design of the Streams Application by considering the requirements that the system has to satisfy. Typically, there are a number of classes of applications that are a good natural fit for the Streams platform. Examples of those application classes are identified and discussed in 3.1.2, “Application purpose and classes of applications” on page 78.

Next, we explore the interfaces of the system, such as what are the inputs that you can use for the application and what are the known outputs that you need to generate. Some of these interfaces are likely to be to existing automated systems, so you should be able to determine the technical details of these interfaces. Some of the inputs and outputs might be new, predicted or unknown, in which case you need to build in flexibility to the application to be able to handle these less well defined interfaces. A more detailed discussion about the design of data sources is in 3.1.3, “Data sources” on page 80, and outputs are further discussed in 3.1.4, “Output” on page 83.

In cases of complex analytic requirements, it may be necessary to enhance the standard Streams Operators by calling out to external software packages or to compile and link the application with existing code. If this is the case, then the design needs to consider the nature of the existing analytics to be used. This topic is discussed further in 3.1.5, “Existing analytics” on page 86.

As Streams is outstanding at delivering high performance, the success of the application relies on identifying, estimating, and forecasting the performance that is required from the application. This topic is covered in 3.1.6, “Performance requirements” on page 87 and includes a discussion about the volumes of data and the processing speed required.

Figure 3-2 graphically summarizes the steps that form the foundation of this initial stage of application design. These steps are:

1. Identify the purpose of the proposed system and determine whether or not it aligns with one of the application classes.
2. Define the inputs and outputs required.
3. Identify existing analytics to use.
4. Assess the performance, time, and reliability requirements.

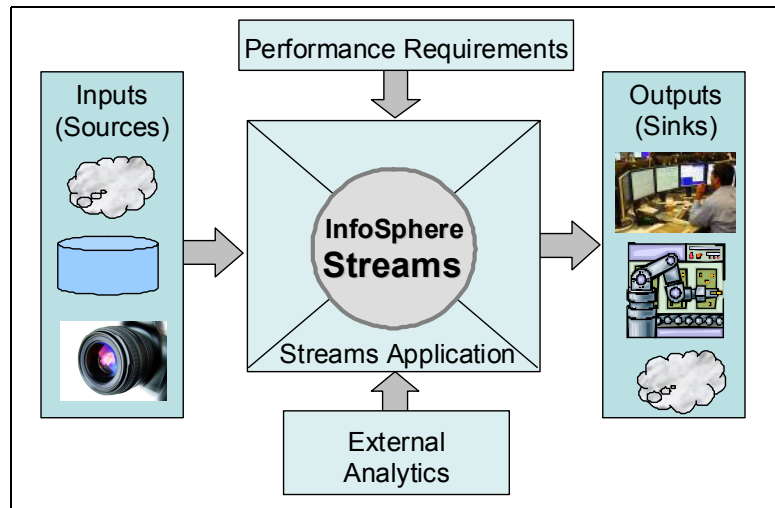


Figure 3-2 Streams Application high level design aspects

3.1.2 Application purpose and classes of applications

The Streams platform was developed with the aim of delivering high throughput, low latency data processing, flexible extension, and integration. The following list contains a number of classes of applications that are highly suitable for the Streams platform:

- *Throughput focused* applications, where extremely large volumes of data need to be processed.

Most Streams Applications involve high throughput to some extent. Some applications are purely focused on performing simple analysis of large streams of data where the sources of the data are well known and will seldom change.

For example, an application performing radio astronomy imaging processes huge streams of data from a fixed number of fixed radio telescopes to allow the production of astronomical images.

The Streams run time achieves high throughput because it seldom persists data, and the run time reduces communication impact by using intelligent grouping of application Operators.

- *Latency focused* applications, where data needs to be processed extremely fast to satisfy the requirements and realize the expected benefits.

In some applications, an opportunity for action needs to be identified from the available data quickly for it to be of value. Any delay in processing will reduce the value of the opportunity, or might result in a missed opportunity altogether.

For example, a financial services application to find profitable stock trading opportunities must handle large volumes of data and must identify trades as soon as possible. If the analysis is slow, then the stock prices might have adjusted to reduce the profit from the trade, or faster competitors might have already seized the opportunity.

Low latency in high volume systems can be achieved by allowing parallel processing of application elements, reducing communication impact between those elements and avoiding the persistence of the data.

- *Discovery focused* applications, where you process data to discover hidden patterns of value.

In this class of applications, you explore data to uncover patterns in the data that are not known or have not yet been discovered. You need to formulate hypothesis about patterns, possibly by processing stored data outside of Streams and then testing the streams of data coming through the application to verify the hypothesis.

For example, fraud detection in the financial and telecommunications industries is an adversarial setting where the fraudulent activities are constantly evolving to overcome the fraud detection. In this case, the hypotheses of the fraud detection algorithms need to constantly adapt to new situations.

Streams has a built-in Data Mining Toolkit that allows Streams Applications to evaluate data according to models that are derived from stored historical data.

- *Prediction focused* applications, where you anticipate events by monitoring trends in the data.

In this situation, you look at trends in data and predicting significant change, which then might require some action be taken by a human.

An example is predicting changes in data from medical sensors. This task can follow trends in physiological signs and can alert a medical practitioner when these signs are thought to require attention.

You can produce a Streams Application to track data from a large number of sensors, correlate the readings from them, and selectively output anomalous behavior as alerts.

- *Control focused* applications, where you monitor and are able to control processes.

You use Streams to measure a well known situation, such as a manufacturing process. The behavior of the situation might be well known and can be predicted by extrapolating the data. In addition, you might be able to control aspects of the process and directly provide feedback to the monitored situation.

The application classes are briefly summarized in Table 3-1, with specific aspects of their design highlighted.

Table 3-1 Application classes and design aspects

Class	Sources	Output	Dependency	Performance
Throughput	Known, Fixed	Analysis		Throughput
Latency	Known, Fixed	Opportunities		Latency
Discovery	Partly known, Dynamic	Novel observation	Hypotheses, Stored data	
Prediction	Known, Multiple	Warnings	Human input	Accuracy
Control	Known, Controllable	Controls, Feedback		Latency

3.1.3 Data sources

Identifying and understanding the data sources available to the Streams Application is an essential aspect of the design. Some might be well known and well defined, and you might need to ensure that you have flexibility to accommodate additional sources when they become available.

Information content

What is the information content of each source? Regardless of the format and structure of the data source, what are the relevant data items that you expect to extract? Here are some examples:

- A feed of video from a traffic camera may allow us to extract information about which vehicles are travelling on a road. The same camera may allow us to identify accidents and alert the relevant authorities to react.

- ▶ Data sources from medical sensors can give us specific readings of a patient's condition.
- ▶ Feeds of data from web logs (blogs) may allow us to identify the reaction to a business announcement and allow us to respond promptly and appropriately.

Method of access

How can you access the data of each source? A data source can be accessed using a number of flexible, configurable options. For example, they might be:

- ▶ File based, for use when data is being deposited on a locally accessible file system.
- ▶ TCP based, so you can access a data stream via a TCP network connection, providing reliable communication of the stream contents from the originator. The Streams Source can be configured to be the server or client side of a TCP connection depending on the requirements.
- ▶ UDP based, so you can access a data stream via a UDP network connection. Unlike TCP communications, UDP does not support reliable message transfer, so it should only be used in situations where it would be acceptable to lose some of that data to gain additional speed of transfer. The Streams Source can only be the server side of a UDP connection, so the system originating the data needs to connect to the Streams UDP socket.

These options provide different levels of reliability. In some cases, reliability might be essential, in other cases missing some Tuples from an input source might be an acceptable alternative. You should consider the following questions to decide which case is appropriate:

- ▶ Is the input data being pushed from a remote system? If you miss the opportunity to observe and consume this data, will it be lost?
- ▶ Is the data is, in some way, buffered or stored so that you can request the contents as and when you are ready?

Variety of sources

How different is the information content of the various sources? Consider the following examples:

- ▶ You might consume a large number of data sources that all contain the same type of data, such as a huge number of temperature sensors from a manufacturing process.

Even if the streams are in different formats, it should be possible in this case to convert to a common format, or extract out the common components, merge the streams, and process the streams in a similar way.

- ▶ You might consume a number of different types of data, such as news reports (textual and TV video), stock market feeds, and weather reports to predict stock price movement.

When the data is of different types, you need more complex comparison, aggregation, and join logic to be able to correlate the streams of data. In this case, it is useful to identify where there are common identities (or Keys, in database terminology) that will allow us to relate one stream to another by using, for example, the built-in join Operators.

Structure of data

How well organized is the information that you receive from the data source and how much of the information is likely to be useful? To help answer that question, let us look at the two primary types of data that are available, structured and unstructured:

- ▶ Structured data is typically well organized. It is likely to have originated from another computer system, and so will be already processed into a refined form. Therefore, the data received will be known and it is possible that the data will be filtered to provide only those data elements of interest.

An example of a structured data source is the feed of data that can be acquired from stock markets. The data will be well defined in terms of the Attributes that you will receive and the meaning of the data (such as the trading prices for particular stocks) will be well understood.

- ▶ Unstructured data is typically less well organized and will need to be identified and extracted from the Source form. Discerning information of relevance could require the use of specialized algorithms that need to be integrated with the Streams Application.

An example of an unstructured data source is the wealth of web logs (blogs) data that is published on the Internet. This data is written in free-form text, can be about a wide range of topics, and might be reliable or unreliable sources of information and opinion. As an example, a market research company might be looking for opinions on products in the blogs of consumers, but will have to process a significant volume of data to enable the extraction of the relevant portions.

What is the data format of the information received from the data source? Streams supports two main formats of data:

- ▶ Alphanumeric: As a default, each line of text is read by the Streams data source and turned into a Tuple. A Tuple is the Streams term for the individual element of data. Think of a Tuple as a row in database terminology, and an object in C++ and Java terminology.
- ▶ Binary: This is the type of data format that comes from video and audio inputs.

Error checking and optional data

How consistent is the information in the Source data stream? In the situation where you have highly structured data, each field in the incoming data will be present, and will have a valid and meaningful value.

However, you might find that the data has values that are omitted from the input, or those values might be invalid. For example, a field that you expect to be a number might be empty, it might include non-numeric values, or it might be a valid number but far larger or smaller than you expected.

You should identify the level of error checking required and specify what to do when errors or omissions are observed:

- ▶ You might need to detect errors in the input stream and either correct or discard these elements.
- ▶ Certain Attributes in the stream Tuples might be optional, and therefore not present in some cases. The appropriate default values should be set in this case.

Control of the data source

In some cases, the data sources might be controllable, that is, they have some mechanism by which you can modify the behavior of the system generating the data. In this case, the results of the Streams Application can be used to affect the data produced.

One example of this is in the manufacturing use cases where you may control the conditions and ingredients of the production process to fine-tune the results and improve the quality of the product. Another example of data source control is in the use of security cameras, where you may control the direction and zoom of a camera to observe an event of interest.

3.1.4 Output

There will be downstream systems and applications that consume the data output by the Streams Application. The Sink Operators of the Streams Application need to be configured and customized to provide the data to these systems in a format suitable for their use. So, you need to identify the downstream systems that will consume the Streams output data.

You need to determine the application programming interfaces (APIs) of the downstream systems:

- ▶ Can the downstream systems read data from files locally accessible from the Streams Runtime? If this is the case, then you can write output files to suitable directories on the Streams file system and the downstream systems can take the data from there.

For high volume data, persistence to disk might be a performance limitation. Therefore, you might need to configure a high speed file system, or send the output using network communication.

- ▶ You can output a data stream over a TCP network connection, providing reliable communication of the stream contents to the recipient. The Streams Sink can be configured to be the server or client side of a TCP connection depending on the requirements of the receiving system.
- ▶ You can output a data stream over a User Datagram Protocol (UDP) network connection. Unlike TCP communication, this does not support reliable message transfer, so use UDP only in cases where some loss of that data is acceptable to gain additional speed of transfer. The Streams Sink can only be the client side of a UDP connection, so the Sink will stream data to a specified IP address and port over a UDP socket.
- ▶ There may be another transfer protocol required to send messages over some specific messaging protocol. In this situation, there could be pre-written adapter Operators available to support these protocols or you may need to produce a user-defined Sink Operator.
- ▶ It could be that you need to use a specific application client, such as a database client. In this situation, there may be pre-written adapter Operators to integrate with these client protocols, or you may need to produce a user-defined Sink Operator.

You need to identify the format and structure of the data that is required by the external system:

- ▶ The required format could be either text-based or binary. Having these options means a different configuration will need to be defined in the Streams Sink Operator.
- ▶ You will need to know the required structure of the data, as well as what data items and data types are required.

The nature of the output is determined by the purpose and requirements of the Streams Application. In the following sections, we provide some examples of different types of output based on the purpose of the application.

Characterization or metadata

If an application is dealing with high volumes of data, one output might be a description that characterizes aspects of the data stream in a more concise representation. This more concise representation can then be used without having to know the details of the original streams.

For example, when receiving sensor readings from health sensors, you can summarize the individual readings into aggregate values, such as the average value, the maximum and minimum, or some statistical measure of the data variation.

As another example, consider a traffic control system that takes feeds from road traffic cameras. This system will be receiving high volumes of binary video data from the camera and can then output a concise description of relevant events, such as the registration number of any identified vehicle, the location, and the time of the event. The original video data may be needed as an audit trail of the processing, but is not actually required for further analysis processing.

Opportunities

The fast processing that can be achieved with Streams provides the possibility that you can identify and act on specific situations, thereby gaining a competitive advantage.

A prime example of opportunity identification is that of algorithmic trading in financial markets. Many areas of the financial services industry analyze larger and larger volumes of data to make their trading decisions. The value of a trading decision is highly time-sensitive, so faster processing of this data can result in making trades at lower prices, which can result in greater profits.

Novel observations

Combining data from different sources and performing statistical analyses of the correlations allows Streams to uncover previously unknown patterns and observations in the data streams.

An example of this situation is using radio astronomy to find gamma ray signals among the data from thousands of telescope antennae.

Warnings and alerts

Streams Applications can be used to monitor a stream of data and highlight when Attributes in the data change state, particularly when an action of some sort should be taken to rectify an error or investigate suspicious activity. An output of warnings and alerts can be generated and fed into downstream systems to prompt action of a suitable nature.

This is the case in the health monitoring example where Streams can be used to perform analysis of medical sensors, bringing significant changes in status to the attention of appropriate medical practitioners for further examination.

Controls and feedback

The Streams Application might be able to directly impact the systems being observed. It can generate outputs that can then be fed as control signals to the automated machines controlling the monitored system, thus providing feedback to the system.

One example of this situation is in the manufacturing use cases where you may control the conditions and ingredients of the production process to fine-tune the results, thereby improving the quality of the product.

Another example of data source control is in the use of security cameras, where you may control the orientation and zoom of a camera to observe an event of interest.

3.1.5 Existing analytics

The standard Operators provided by Streams provide a useful and wide ranging toolkit for assembling applications to deliver stream computing systems. However, in cases of complex analytic requirements, it might be more effective to enhance the standard Streams Operators by invoking external software packages or by compiling and linking the application with other existing analytic applications. If this is the case, then the design needs to consider the nature of these external packages or applications to ensure that they are consistent with the requirements and well integrated.

For example, when dealing with unstructured data, you might utilize specialized analysis systems, such as video processing, audio processing, natural language processing, or language translation systems to analyze the incoming data stream and extract out items of relevance. The relevant metadata can then be used as a data stream within the rest of the Streams Application.

Another example is the wrapping of existing coded algorithms to process streams data. When migrating to Streams from a traditional platform, algorithms for data processing are likely to exist. Code developed in C++ and Java can both be linked into Streams as user-defined Operators (UDOPs), allowing streams to perform these algorithms in high volumes, and combine them with other sources of data.

You need to consider the implications of the new Streams Application on any truly external systems. The Streams Application will generate extra load on the system and you should validate that this does not cause problems.

The design needs to determine what you should do if the external system fails or is unavailable. Compensation actions may be required, such as skipping a section of processing or alerting a human operator about the problem.

3.1.6 Performance requirements

One of the main goals of Streams is to deliver high throughput and low latency computation. To achieve this goal, the performance requirements of the system need to be understood. In this section, we discuss a number of aspects of the performance requirements that should be addressed.

You should determine the expected volumes for each data source, and consider the following measures:

- ▶ What is the average throughput over a sustained period of time?
- ▶ What is the peak expected throughput in any short period of time?
- ▶ What growth in data volumes is anticipated in the future?

Knowing the average and peak throughput, for example, allows you to determine the power and quantity of physical hosts required by your target application. Sizing of the physical environment is discussed further in 4.3, “Sizing the environment” on page 136.

You need to determine an acceptable alternative when the volume of data from a data source exceeds what the system can handle. Some alternatives are:

- ▶ You can discard excess data. The Source and associated Operators can be designed to act as a throttle, filtering the incoming data to a specific rate. It may also be possible to combine multiple records into aggregates (as so reducing the number of records processed downstream), or to discard excess Tuples, either by a random process of sampling or by identifying Tuples of lower potential value.
- ▶ You can buffer excess data in the expectation that this is a peak load period and as the load declines, the system will be able to process that excess data. Note that using this approach will increase the average latency (time to process each individual entry) as latency will also include the time that a Tuple is buffered and waiting to be processed. As such, this approach might be undesirable in an extreme low latency focused system.

Extreme performance requirements might lead you to segment the application to enable processing of smaller chunks, which may then be deployed to separate Hosts and processed in parallel. The impact of sizing and extreme volumes on the application structure is discussed further in 4.4, “Deployment implications for application design” on page 137.

3.2 Streams design patterns

In 3.1.2, “Application purpose and classes of applications” on page 78, we identify a number of application classes representing the overall purpose of a system and its overriding design characteristics. In this section, we identify and detail a set of design patterns that have been encountered in Streams Applications.

Design patterns are a formalization of common structures and interactions that arise repeatedly from the process of designing a range of applications. The approach of identifying design patterns and documenting them in a standard format is a way to simplify the design process. An understanding of these common design patterns should enable you to use them as building blocks for your particular design. The design of an application can then be constructed as a combination of design patterns rather than a combination of individual Operators. This is easier to understand and means that you are able to benefit from the distilled design wisdom of previous successful applications.

As implementations of Streams grow, you should expect the number and types of design patterns to also grow. The following design patterns have been identified at this time:

- ▶ *Filter pattern* for data reduction
- ▶ *Outlier pattern* for data classification and outlier detection
- ▶ *Parallel pattern* for high volume data transformation
- ▶ *Pipeline pattern* for high volume data transformation
- ▶ *Alerting pattern* for real-time decision making and alerting
- ▶ *Enrichment pattern* for supplementing data
- ▶ *Unstructured Data pattern* for supporting unstructured data analysis
- ▶ *Consolidation pattern* for combining multiple sources
- ▶ *Merge Pattern* for combining similar sources
- ▶ *Integration Pattern* for leveraging existing analytics

The following sections describe each of these different design patterns. We outline the purpose and benefits associated with each pattern, give examples of their use and present a Stereotype, which is a simplified illustration of the pattern constructed from the standard Streams Operators.

3.2.1 Filter pattern: Data reduction

When you are processing large volumes of data, a large proportion of the data may be considered to be irrelevant to your analysis. For example, you may only be interested in infrequent, unusual occurrences within the data streams.

In this case, a Streams Application can be used to filter the data, reducing it to the items of significance. The application might analyze the data stream, identify the elements of interest, and create a new stream of data from the significant items. With this new stream, a reduced number of these valuable parts will be processed by the remainder of the Streams Application.

Figure 3-3 shows the principle of the Filter pattern. The application reduces the volume of data so that the output is more manageable.

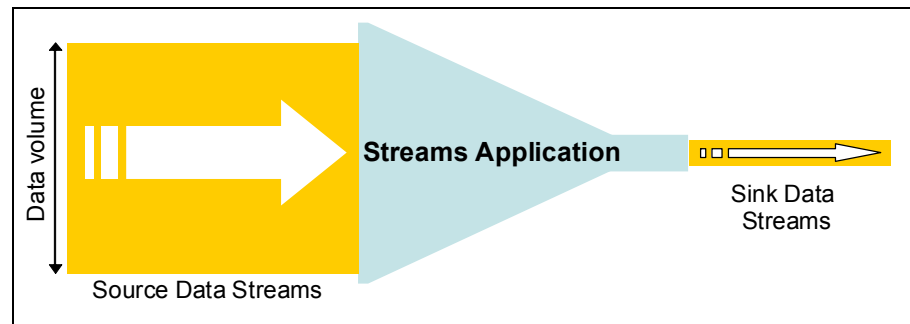


Figure 3-3 Effect of a filter on Streams volumes

Example

In financial services, you might be interested in price changes for stocks in your stock portfolio. You can consume a feed of data from the stock market and use a filter to reduce this feed down to only the stocks of interest to you.

In a health care situation, a large volume of readings will be generated from medical sensors. These will usually stay within an expected range, but when they fall outside this range, an alert needs to be raised. A filter can be used to detect the dangerous sensor readings.

Stereotype

A simple implementation of the Filter design pattern illustrating value identification and extraction is shown in Figure 3-4. In this illustration, a Source Operator reads in the data stream and a Functor Operator is used to determine the Tuples of interest and only passes these on to the Sink Operator, which sends these to the relevant recipients.

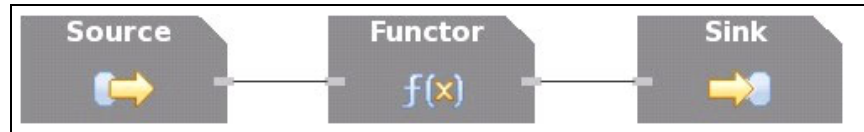


Figure 3-4 Application topology of the filter pattern stereotype

Stereotype walkthrough

Here are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the `input.dat` file, creating the `unfilteredInput` stream.
2. A Functor Operator processes the `unfilteredInput` stream and determines the Tuples of interest, in this case those strings that are greater than the string `Interesting Data`. The output stream from the Functor is the `FilteredOutput` stream and only contains the Tuples of interest.
3. A Sink Operator takes the `FilteredOutput` stream and outputs the data to a file named `output.dat`; this data will be used by relevant recipients.

Variations

There might be variations in the process, but they may be handled by the following types of options:

- ▶ The filtering logic offered by the standard Functor Operator can be extended by adding user defined logic, allowing the maintenance of state variables and the use of control logic (such as loops and if-else conditions).
- ▶ For more complex analysis, the Functor Operator may be expanded into multiple Operators, processing the stream in multiple serial or parallel steps.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-1.

Example 3-1 Streams Application code of the filter pattern stereotype

```
[Application]
  Filter
[Program]
```

```

vstream Schema (field: String)

stream UnfilteredInput (schemafor(Schema))
  := Source ()
    ["file:///input.dat", csvformat, nodeays]
    {}

stream FilteredOutput (schemafor(Schema))
  := Functor (UnfilteredInput)
    [field = "Interesting Data"]
    {}

Nil := Sink (FilteredOutput)
    ["file:///output.dat", csvformat, nodeays]
    {}

```

Suggestions

When processing high volume streams, you should consider filtering the streams to reduce their volumes as soon as possible after they are ingested. This can reduce the total processing workload of the Application, allowing a higher volume of throughput, with lower latency and potentially using less hardware.

Figure 3-5 shows two contrasting filtering approaches. The top diagram shows a Streams Application that filters the Source streams early on in its processing and the bottom diagram shows a Streams Application that filters the Source streams later on in its processing.

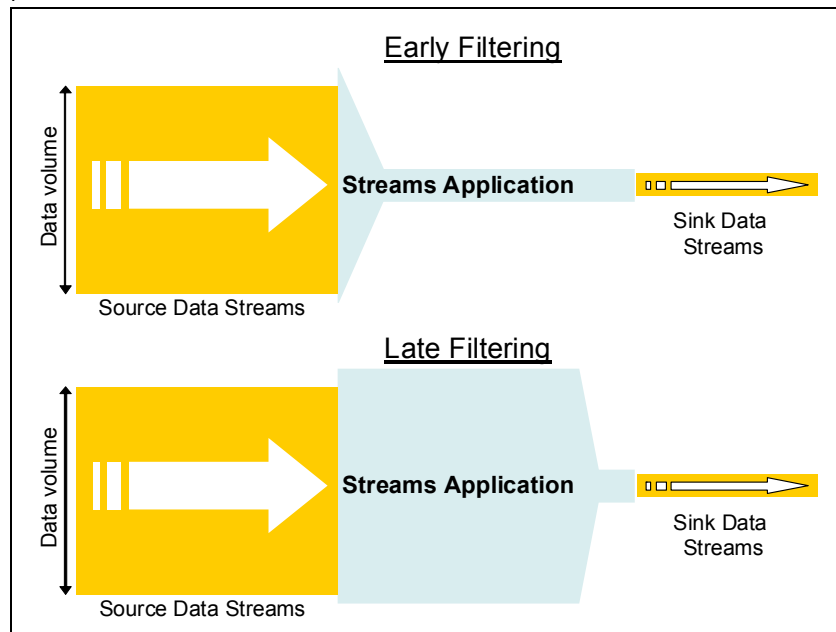


Figure 3-5 Effect of early and late filtering on Streams volumes

In these figures, the area depicted as the Streams Application is proportional to the amount of processing that the application needs to perform. In the early filtering case, most of the Streams Application only has a small number of data Tuples to process, so the shaded area is relatively small. In the late filtering case, most of the Streams Application has a large number of data Tuples to process, so that area is far larger, indicating a greater processing workload for the application.

3.2.2 Outlier pattern: Data classification and outlier detection

You can identify significant situations in a real scenario by monitoring the situation and detecting data that deviates from the usual patterns.

You analyze the received data to understand the normal behavior for that particular stream at that time. As you continue, most values received will fall within the normal range and so do not need to be processed. Here you are only

interested in processing the outliers, which are the rare values that are considerably larger or smaller than usual.

Figure 3-6 illustrates the concept of statistical outliers. The majority of data items are found within a normal range or variability. However, there are a number of data items outside the normal range that are deemed to be outliers.

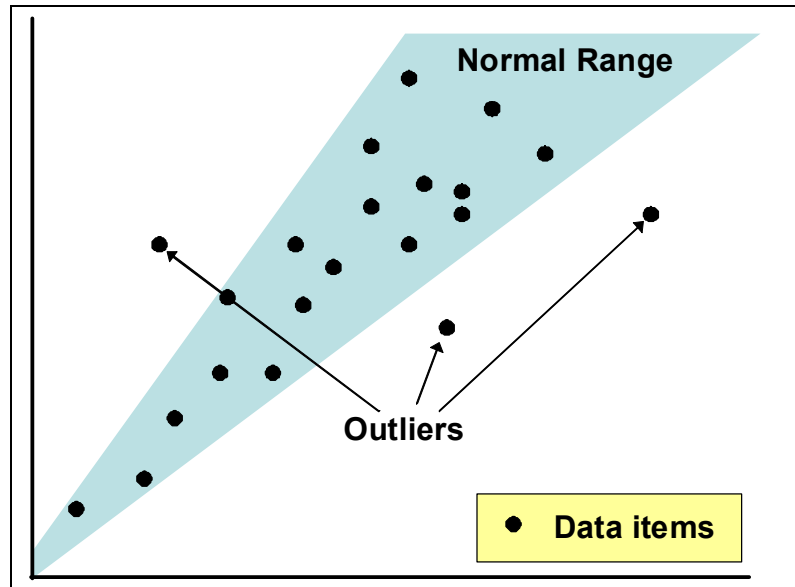


Figure 3-6 Statistics outliers in a body of data

In this case, a Streams Application can keep a window of recent values for each group in the stream and analyze this window to classify the normal behavior. Each entry received can be compared with the classification of recent values. Significant entries are passed through in a new stream of data and subsequent Operators can process only these significant entries.

Example

One aspect of the financial services use case is the requirement to monitor stock prices, as you are interested in significant changes in the price. You can consume a feed of market activity data from the stock market, determine the average price for that period of time, and then identify outliers within the data when individual prices are especially higher or lower than the average. These outliers highlight the potential for profitable stock trades.

When processing video streams from security cameras, you are interested in changes in the images, such as people and vehicles moving or objects appearing and disappearing. Video processing algorithms can characterize the background image and can detect when the images change. Significant events may be passed as alerts to the responsible security guard.

Stereotype

A simple implementation of the Outliers design pattern, illustrating data classification and outlier detection, is shown in Figure 3-7. In this illustration, a Source Operator reads in the data stream, and an aggregate determines the average for recent values of the data values. The join compares the latest value of a particular type with the average and only passes through the values that are unusually high or low. The Sink Operator then passes these to the relevant recipients.

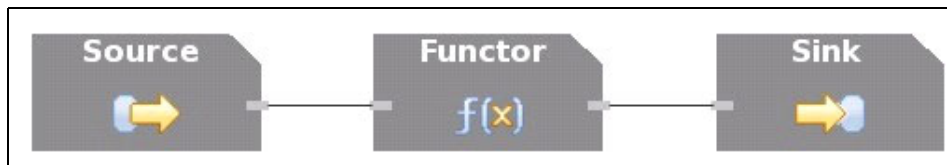


Figure 3-7 Application topology of the Outliers pattern stereotype

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the `input.dat` file, creating the `RawValues` stream.
2. An aggregate Operator determines the average of the last 10 Instances of the value `Attribute` for each Source ID. The output stream from the aggregate is the `ValueCharacterisation` stream containing the average values for each `sourceID`.
3. A join Operator takes individual Tuples from the `RawValues` stream and compares these with the latest average for that Source ID, which it gets from the `ValueCharacterization` stream.

The condition in the Operator only allows values that are greater than twice the average, or less than half the average to get through. The output stream from the join is the stream `OutlyingValues`, containing only these larger or smaller than average values.

4. A Sink Operator takes the `OutlyingValues` stream and outputs the data to a file named `output.dat`, and this data will be used by the relevant recipients.

Variations

There might be variations in the process, but they may be handled by using the following types of options:

- ▶ The aggregate and join Operator can operate on the entire stream of data as a single set of data or can group the data by one of the Attributes. For example, when processing health monitoring data for many patients, you need to consider all the data for each patient individually, so you group the data by the patient ID.
- ▶ The categorization of the data may be determined by a more complex algorithm than can be supported in an aggregate function. A set of Operators, possibly user-defined Operators, may be used. A call to an external system may be required to determine interesting behavior in unstructured data.
- ▶ If the model for data classification relies on a large body of historical data, the data mining techniques can be applied outside the Streams environment to determine the classification or scoring model.

The data mining model can then be implemented in a Streams Application, which can score the items in real time and determine the items that are significant.

Streams comes with a Data Mining Toolkit that provides support for data mining classification and scoring based on the Predictive Model Markup Language (PMML) standard. This provides interoperability with IBM InfoSphere Data Warehouse and other standards-compatible data mining packages.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-2.

Example 3-2 Streams Application code of the Outliers pattern stereotype

[Application]

Outliers

[Program]

```
vstream Schema (sourceID: String, value: Double)
```

```
stream RawValues (schemafor(Schema))
```

```
:= Source ()
```

```
["file:///input.dat", csvformat, nodelays]  
{}
```

```
stream ValueCharacterisation (sourceID: String, average: Double)
```

```
:= Aggregate (RawValues<count(10), count(1), perGroup>)
```

```
[sourceID]
```

```

    { Any(sourceID), Avg(value) }

stream OutlyingValues (sourceID: String, value: Double, average:
Double)
    := Join (RawValues<count(0)> ; ValueCharacterisation<count(1),
perGroup> )
    [ {sourceID} = {sourceID},
      ((value > 2.0d * average) |
       (value < 0.5d * average)) ]
    {$1.sourceID, value, average}

Nil := Sink (OutlyingValues)
      ["file:///output.dat", csvformat, nodeays]
      {}

```

Suggestions

Consider applying this pattern as soon as possible after data is ingested. The pattern results in a significant reduction in the data being analyzed by the Streams Application, reducing the total processing workload of the application and allowing a higher overall volume of throughput.

3.2.3 Parallel pattern: High volume data transformation

For high volume streams, you might find that a data transformation activity cannot be efficiently processed on a single Host computer and you will need to spread the workload across multiple Hosts. A failure to do this task could limit the throughput that the Streams system could support for this application.

Figure 3-8 shows a Streams Operator that is limiting the throughput of the application. The Operator running on this Host will take 0.0001 seconds to process a Tuple. You can estimate the maximum throughput of this Operator as 10,000 Tuples per second ($1/0.0001 = 10,000$). If we assume the requirement is to process 15,000 Tuples per second, it cannot be achieved with this deployment. The Operator will be a bottleneck, and impose a limit on the performance.

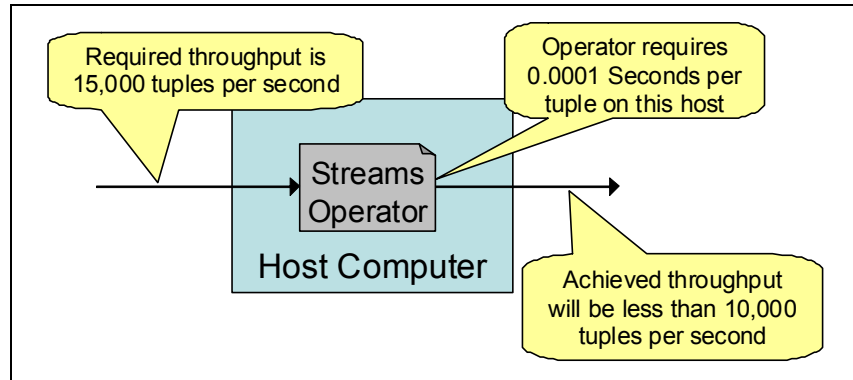


Figure 3-8 Streams Operator limiting the application throughput

The workload could be segmented in a number of ways, such as:

- ▶ Segmenting the stream data into multiple substreams and processing them in parallel.
- ▶ Dividing the transformation activity into multiple serial stages, which can be processed sequentially, and then processing subsequent stages on different Hosts.

The determination of which approach should be taken is discussed in more detail in 4.4.2, "Overcoming performance bottleneck Operators" on page 140.

Figure 3-9 illustrates the throughput improvement from adopting the parallel pattern. Here the same Operator is running on two separate Host computers. Each Operator will have a throughput of around 10,000 Tuples per second, giving a total of around 20,000 Tuples per second. Because the requirement is to process 15,000 Tuples per second, it can be achieved with this deployment configuration.

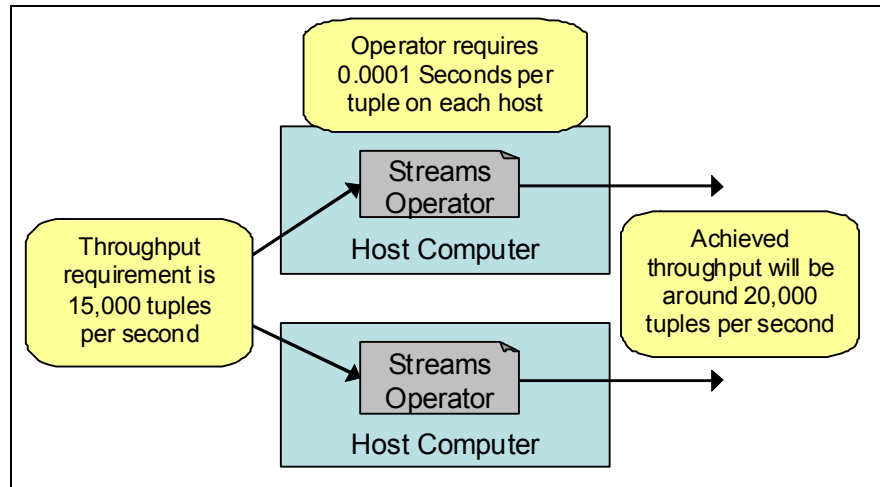


Figure 3-9 Throughput improvement with Parallel Operators

Example

The processing of video and audio streams tends to be processor intensive. In this case, video streams could be segmented and distributed across a cluster of servers. The results of the processing could then be merged back into a single stream of results. A simple implementation of the Parallel design pattern is shown in Figure 3-10.

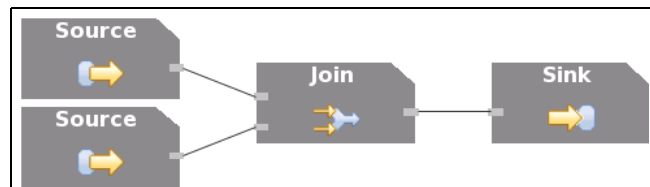


Figure 3-10 Application topology of the Parallel pattern stereotype

Stereotype

In this illustration, a Source Operator reads in the data stream and a split Operator acts as a load balancer and segments the incoming stream into five substreams. Each of the substreams is processed in the same way by a Functor

Operator on a different Host computer. The Sink Operator then consumes all the output streams.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream, from the `input.dat` file, creating the `RawValues` stream.
2. A Split Operator segments the incoming stream into five substreams (because `number_parallel` is defined as 5) named `ValuesPartition1` to `ValuesPartition5`. The streams are segmented based on the `sourceID` Attribute, so all Tuples with the same `sourceID` will be sent to the same output stream.

Note: The preprocessor instructions `for_begin .. for_end` expand out to form the definitions of multiple output streams and multiple Functor Operators.

3. Each of the segmented streams is processed in the same way by a Functor Operator; in this case, the result is calculated as the square of the input value. Each Functor outputs one of the streams `ValuesProcessed1` to `ValuesProcessed5`. Each Functor can be deployed on a different Host computer, spreading the processing load.
4. The `ValuesProcessed1` to `ValuesProcessed5` streams are defined together into a stream bundle called `ProcessedBundle`. A Sink Operator takes the `ProcessedBundle` streams and outputs the data to a file named `output.dat`; this data will be used by relevant recipients.

Variations

There might be variations in the process, but they may be handled by the following option.

In the Parallel pattern, it is possible to replace the Split Operator with a set of Functor Operators, each of which filters the Source stream to select the specific Instances of interest.

This variation is illustrated in Figure 3-11. Here the shaded boxes indicate the Hosts on which the Operators are deployed. You can see that to minimize inter-host communication, the first set of Functors have been collocated with the source. This is because the first set of Functors after the source receives all the Tuples from the Source Operator.

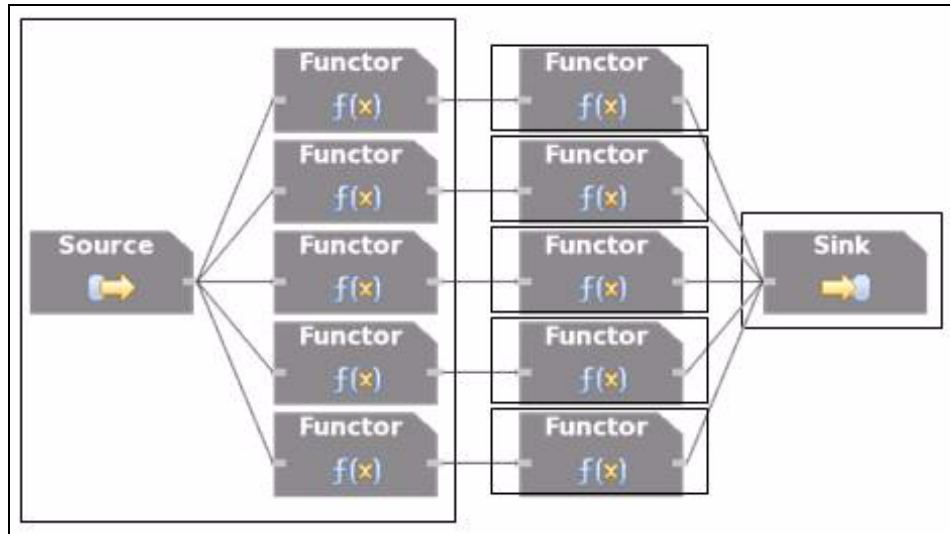


Figure 3-11 Variation of the Parallel pattern with no splitter

Some sample code that illustrates how that implementation might be supported is shown in Example 3-3.

Example 3-3 Streams Application code of the Parallel pattern stereotype

```
[Application]
  Parallel
[Program]
#define number_parallel 5

vstream Schema (sourceID: Integer, value: Double)

stream RawValues (schemafor(Schema))
  := Source ()
    ["file:///input.dat", csvformat, nodeays]
    {}

for_begin @L 1 to number_parallel
stream ValuesPartition@L (schemafor(Schema))
for_end
```

```

:= Split (RawValues)
[sourceID]
{}

bundle ProcessedBundle := ()

for_begin @L 1 to number_parallel
stream ValuesProcessed@L (schemafor(Schema), result: Double)
:= Functor (ValuesPartition@L)
[]
{result := value * value}

ProcessedBundle += ValuesProcessed@L
for_end

Null := Sink ( ProcessedBundle [:] )
["file:///output.dat", csvformat, nodeays]
{}

```

Suggestions

Use of the Parallel pattern to improve throughput could be used when considered necessary by the volumetric requirements and the sizing of the hardware environments. This topic is discussed in more detail in 4.4, “Deployment implications for application design” on page 137.

3.2.4 Pipeline pattern: High volume data transformation

For high volume streams, you might find that a data transformation activity cannot be efficiently processed on a single Host and you need to segment the workload across multiple Hosts.

One way to spread the workload is to divide the transformation activity into multiple serial stages that can be performed sequentially. Each of these steps can be performed on different Host computers.

Figure 3-12 illustrates throughput improvement from adopting the Pipeline pattern. A unit of processing has been segmented into three Operators, with each Operator running on two separate Host computers. The overall throughput of the pipe is determined by the slowest Operator, which in this case gives a throughput of around 20,000 Tuples per second ($1/0.00005$). If the requirement is to process 15,000 Tuples per second, it may be achieved with this deployment configuration.

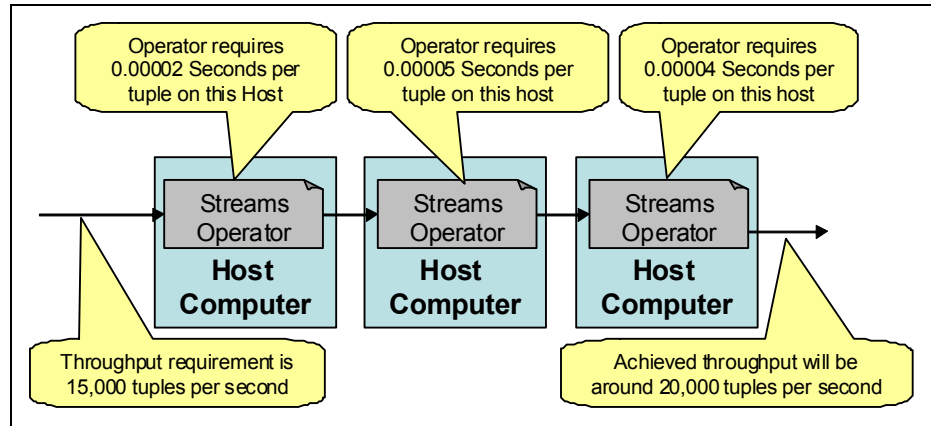


Figure 3-12 Throughput improvement with Operators in pipeline

Example

Streams Applications are usually well divided because they are constructed by combining small, granular Operators. If you include a significant amount of user-defined logic in a single Operator (which could be a Functor, UDF, UDOP, or UBOP), then this could become a bottleneck in a high volume system. The user-defined logic may be refactored by segmenting it into a number of lesser Operators. The Streams Application could then be able to deploy the Operators across multiple Hosts and achieve better throughput.

A simple implementation of the Pipeline design pattern is shown in Figure 3-13.



Figure 3-13 Application topology of the Pipeline pattern stereotype

Stereotype

Assume a segment of a complicated multi-step user-defined logic has been segmented across three Functors with each Functor performing a small part of the overall processing. The Sink Operator then consumes the final output

stream. The Streams Scheduler is able to perform these separate Functors on separate Hosts, spreading the processing workload.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the `input.dat` file, creating the `RawValues` stream.
2. A pipeline of Functors process the stream of data, creating `ValuesStage1`, `ValuesStage2` and `ValuesStage3` streams. At each stage, the value is doubles. A `2d` in the code indicates that 2 is a value of *Double* type.
3. A Sink Operator takes the `ValuesStage3` stream and outputs the data to a file named `output.dat`; this data will be used by relevant recipients.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-4.

Example 3-4 Streams Application code of the Pipeline pattern stereotype

```
[Application]
  Pipeline
[Program]

vstream Schema (sourceID: Integer, value: Double)

stream RawValues (schemafor(Schema))
  := Source ()
    ["file:///input.dat", csvformat, nodelays]
    {}

stream ValuesStage1 (schemafor(Schema))
  := Functor (RawValues)
    []
    {value := value * 2d}

stream ValuesStage2 (schemafor(Schema))
  := Functor (ValuesStage1)
    []
    {value := value * 2d}

stream ValuesStage3 (schemafor(Schema))
  := Functor (ValuesStage2)
    []
    {value := value * 2d}
```

```
Null := Sink ( ValuesStage3 )  
        ["file:///output.dat", csvformat, nodelays]  
        {}
```

Suggestions

Use of the Pipeline pattern to improve throughput could be used when considered necessary by the volumetric requirements and the sizing of the hardware environments.

There is a time impact associated with segmenting the processing into multiple stages, so this might not be a preferred option in low latency situations. This topic is discussed in more detail in 4.4, “Deployment implications for application design” on page 137.

3.2.5 Alerting pattern: Real-time decision making and alerting

In a number of situations, it is beneficial to act quickly to changes in data. In these cases, the cost involved in equipment failure may be reduced, or the chance of taking advantage of an opportunity might improve if you react faster to the events.

A Streams Application will process events immediately as they happen, invoking a low cost of data transmission and avoiding data storage during the processing. Analysis of the event data in the streams allow us to determine situations of interest from the huge bulk of data ingested and you can react quickly, either by alerting a human user or by prompting an automated system to respond to the situation.

Examples

In a manufacturing situation, there are multiple sensors in place detecting characteristics of the manufactured products. When an error occurs, or the quality of the product deteriorates, the manufacturing process needs to be adjusted. It may also be necessary to discard some of the product until the quality returns to acceptable levels. Rather than waiting for minutes or hours to respond, Streams Applications should allow processes to be adjusted promptly, resulting in a reduced cost of wastage.

When monitoring health care devices, any anomalous readings should be immediately passed as an alert to the health care provider, allowing them to act immediately and increase the chances of a successful intervention if required.

An application assessing opportunities to trade in company stocks needs to react quickly. The opportunity to trade may only last for a short period of time before prices adjust or faster competitors take advantage of the best trades.

A simple implementation of the Alerting design pattern is shown in Figure 3-14.



Figure 3-14 Application topology of the Alerting pattern stereotype

Stereotype

In this illustration, a Source Operator reads in the data stream. A Functor implements logic to identify conditions to alert. The Sink Operator then passes these to the relevant recipients to act upon.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the input .dat file, creating the RawInput stream.
2. A Functor Operator processes the RawInput stream, determines whether the received Tuple is worthy of an alert (in this case, is the field string equal to *Alert*). The output stream from the Functor is the Alerts stream and only contains the Tuples that need to be raised as alerts.

A Sink Operator takes the Alerts stream and outputs the data to a file named output.dat, and this data will be used by the relevant recipients.

Variations

Be aware that the logic in the decision making process is likely to involve more than a single Operator, or may require user-defined Operators.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-5.

Example 3-5 Streams Application code of the Alerting pattern stereotype

```
[Application]
  Alerting
[Program]
```

```
vstream Schema (field: String)
```

```

stream RawInput (schemafor(Schema))
  := Source ()
    ["file:///input.dat", csvformat, nodelays]
    {}

stream Alerts (schemafor(Schema))
  := Functor (RawInput)
    [field = "Alert"]
    {}

Nil := Sink (Alerts)
    ["file:///output.dat", csvformat, nodelays]
    {}

```

Suggestions

Avoid use of large tumbling windows in the same processing chain as an Alerting pattern, where there is the need for low latency altering. Tumbling windows will only output results when their windows fill up, and this could introduce an undesirable delay in the application.

3.2.6 Enrichment pattern: Supplementing data

In some cases, you find that a stream of data includes a reference to another item of data, but does not include all the details of the item that you need. Details of the referenced item may exist as static or slowly changing reference data, or may be available in a stream from a different data source.

In this case, Streams provides the ability to enhance the data from the stream. The approach is to read in the referenced data and then join streams by matching the reference Attributes, or by specifying a matching condition on a number of Attributes.

Example

A stream of data containing customer purchases may include a Customer ID but will not include the details of the customer, such as their address or age. If you want to determine the total customer spend by location or age, then you need to enrich the order data with customer details that you hold separately. A simple implementation of the Enrichment design pattern is shown in Figure 3-15.

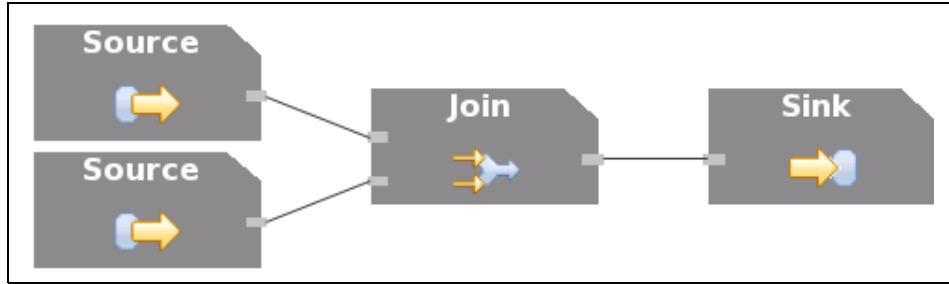


Figure 3-15 Application topology of the Enrichment pattern stereotype

Stereotype

In this illustration, a Source Operator reads in a number of data streams, one of which will be the *actuating* stream, which provides Tuples of data to enrich, and the other will be the *reference stream*, which provides the enriching data.

The join Operator can be configured with a window to store the latest reference data, which could match the actuating stream. When a Tuple is received in the actuating stream, it will match the actuating stream with the reference stream and output any matches. The Sink Operator then passes these to the relevant recipients.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the `input1.dat` file, creating the `RawValues` stream with an ID and a value Attribute.
2. Another Source Operator reads in the data stream from the `input2.dat` file, creating the `EnrichingValues` stream with an ID and a different value Attribute.
3. A join Operator takes the `RawValues` and `EnrichingValues` streams. The `EnrichingValues` stream is processed and the latest *value2* for each Source ID is kept in a window by the join Operator. This behavior is determined by the `<count(1), perGroup>` window definition. Each time a Tuple is received in the `RawValues` Stream, its `sourceID` is matched with the `EnrichingValues` window and a Tuple is output to the `ValueEnrichment` Stream containing the *value1* from the `RawValues` and the latest *value2* from the `EnrichingValues`.

4. A Sink Operator takes the ValueEnrichment stream and outputs the data to a file named `output.dat`; this data will be used by relevant recipients.

Variations

When the amount of reference data gets large, it may be impractical to keep it in a join Operator. You can call out to an external database that holds the data, which can be a persistent disk based database such as IBM DB2®, or, for speed, a persistent in-memory database, such as IBM solidDB.

Streams provides an ODBC enrichment Operator that provides enrichment from DB2, Informix, Oracle, and solidDB via the ODBC standard. A simple implementation of the Enrichment design pattern is shown in Figure 3-16.



Figure 3-16 Enrichment from an external database

Some sample code that illustrates how that implementation might be supported is shown in Example 3-6.

Example 3-6 Streams Application code of the Enrichment pattern stereotype

```
[Application]
  Enrichment
[Program]

vstream Schema1 (sourceID: String, value1: Double)

vstream Schema2 (sourceID: String, value2: Double)

stream RawValues (schemafor(Schema1))
  := Source ()
    ["file:///input1.dat", csvformat, nodelays]
    {}

stream EnrichingValues (schemafor(Schema2))
  := Source ()
    ["file:///input2.dat", csvformat, nodelays]
    {}

# Here everytime a RawValues tuple comes in, it will be enriched with
value2 from the latest
```

```
# the latest EnrichingValues tuple with the same sourceID
stream ValueEnrichment (sourceID: String, value1: Double, value2:
Double)
  := Join (RawValues<count(0)> ; EnrichingValues<count(1), perGroup>)
  [ {sourceID} = {sourceID} ]
  {$1.sourceID, value1, value2}

Nil := Sink (ValueEnrichment)
      ["file:///output.dat", csvformat, nodelays]
      {}
```

Suggestions

Enrichment is likely to be an expensive operation, involving either storing a body of detailed reference data or involving a call to an external system that contains the details. As such, enrichment should take place after the Source streams have been reduced by filtering as much as possible.

3.2.7 Unstructured Data pattern: Unstructured data analysis

You may find that the data sources that you have identified for your application contain the following types of unstructured data:

- ▶ Free form text
- ▶ Video streams
- ▶ Audio streams

The analysis of these sources can rely on complex algorithms, which have already been developed and deployed. These complex algorithms will tend to be processor intensive, increasing the need for parallel processing to achieve a high level of throughput.

This situation can be supported in Streams by using a Functor or user-defined Operator to integrate the existing logic, while segmenting the streams to allow parallel execution of the processor intensive logic on a cluster of Hosts. Streams allows sources that ingest binary data to support video and audio formats.

Example

When processing video data from a security camera, there are existing packages that process the video images, determine foreground movement from the background, and detect differences in images over time.

It is possible to ingest large quantities of unstructured text from web sources, for example, news feeds and blogs. Natural language processing products exist that allow this data to be analyzed, to extract details of people, places, events, and so on, from the text and to produce results identifying these items. Additionally, it may be necessary to perform language translation when processing these unstructured text streams. A simple implementation of the Unstructured Data design pattern is shown in Figure 3-17.

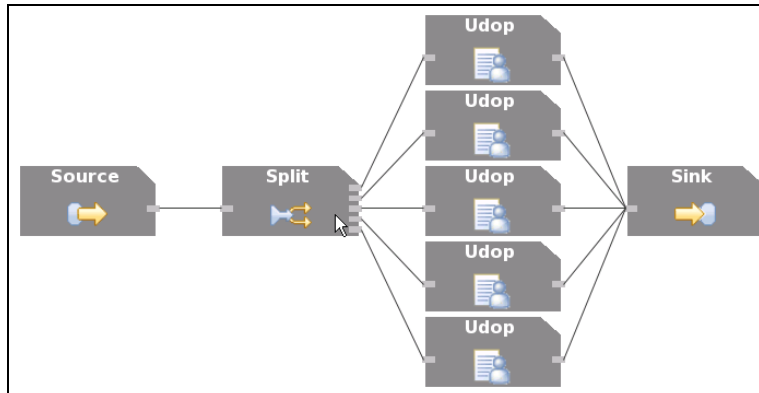


Figure 3-17 Application topology of the Unstructured Data pattern stereotype

Stereotype

In this illustration, a Source Operator reads in the data stream and the stream is segmented into multiple partitions to distribute the load across multiple Hosts. Each Host processes the unstructured data in a user-defined Operator (UDOP) that can call to an external algorithm to process the stream. Either the unstructured data is transformed, augmented, or some meaningful description of the data is extracted. The Sink Operator consumes the UDOP output streams and passes these to the relevant recipients.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream, from the `input.dat` file, creating the RawText stream.
2. A Operator segments the incoming stream into five substreams (because `number_parallel` is defined as 5) named TextPartition1 to TextPartition5. The streams are segmented based on the `sourceID` Attribute, so all Tuples with the same `sourceID` will be sent to the same output stream.

Note: The preprocessor instructions `for_begin` to `for_end` expand out to form the definitions of multiple output streams and multiple Functor Operators.

3. Each of the segmented streams is processed in the same way by a user-defined Operator (UDOP) called *ProcessUnstructuredText*. This UDOP can call an external package to process the *FreeformText* and return a string of *TextMetaData*, summarizing any text of interest in the *FreeformText* Attribute. Each UDOP outputs one of the streams *TextProcessed1* to *TextProcessed5*. Each UDOP can be deployed on a different Host computer, spreading the processing load as the text processing in this case is processor intensive.
4. The *TextProcessed1* to *TextProcessed5* streams are defined together into a stream bundle called *ProcessedBundle*. A Sink Operator takes the *ProcessedBundle* streams and outputs the data to a file named *output.dat*, which will be used by the relevant recipients.

Variations

You can convert the UDOP into a UBOP, which allows more flexibility and reusability of the user-defined logic.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-7.

Example 3-7 Streams Application code of the Unstructured Data pattern stereotype

```
[Application]
  Unstructured
[Program]
#define number_parallel 5

vstream Schema (sourceID : Integer, FreeformText : String)

stream RawText (schemafor(Schema))
:= Source ()
    ["file:///input.dat", csvformat, nodelays]
    {}

for_begin @L 1 to number_parallel
stream TextPartition@L (schemafor(Schema))
for_end
:= Split (RawText)
[sourceID]
```

```

    {}

    bundle ProcessedBundle := ()

    for_begin @L 1 to number_parallel
    stream TextProcessed@L (sourceID: String, TextMetaData: String)
        := Udop (TextsPartition@L)
        ["ProcessUnstructuredText"]
        {}

        ProcessedBundle += TextProcessed@L
    for_end

    Null := Sink ( ProcessedBundle [:] )
        ["file:///output.dat", csvformat, nodelays]
        {}

```

Suggestions

Processing unstructured data is likely to be an expensive operation. As such, it should probably take place after the Source streams have been reduced by filtering as much as possible.

3.2.8 Consolidation pattern: Combining multiple streams

A significant strength of Streams is the ability to consume multiple streams and to combine the information that they contain to reach new conclusions. A business often has many sources of information and typically processes these individually. However, there is more value in being able to consolidate the entire body of information to gain new insights.

Streams Applications may contain a join Operator, which is a flexible and powerful way to combine multiple streams. You can specify different windowing schemes to change the nature of the data combined and you can change the join condition to specify which data is combined.

Example

The example of stock market trading illustrates the value of consolidation. The main stream of stock trading prices can be consolidated with other sources of information that may affect those prices, such as short term and long term weather forecasts or breaking news stories that may strengthen or weaken the position of a company.

These consolidating streams of data can be used to predict likely changes in stock price, allowing us to identify a wider range of profitable trading opportunities. A simple implementation of the Consolidation design pattern is shown in Figure 3-18.

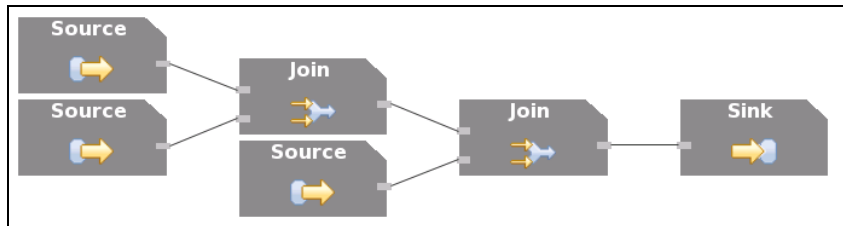


Figure 3-18 Application topology of the Consolidation pattern stereotype

Stereotype

In this illustration, Source Operators reads in the data streams, which are then combined together using the join Operators. The join Operators will have windows defined so that they will keep a set of data Tuples to allow data in one stream to potentially match the data in another stream, in which case their data may be consolidated. The Sink Operator then passes these to the relevant recipients.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. Source Operators reads in the data streams from the input files, creating the RawValues, ConsolidatingValues, and ConsolidatingValues2 streams with IDs and value Attributes.
2. Join Operators take the input streams and combine the values in the streams by matching the sourceIDs, producing the Consolidation1 and Consolidation2 streams. Each join Operator keeps a window of the latest value for each sourceID from both input streams. This behavior is determined by the `<count(1), perGroup>` window definition.

Note: The standard Streams join Operator combines exactly two input streams, so two join Operators are required to consolidate three different streams of data as shown in the stereotype.

- A Sink Operator takes the Consolidation2 stream and outputs the data to a file named `output.dat`; this data will be used by relevant recipients.

Variations

Consolidation is similar to enrichment, which is described in 3.2.6, “Enrichment pattern: Supplementing data” on page 106. The difference is that the Consolidation pattern can generate output Tuples if either of the input streams changes, whereas the Enrichment pattern generates output Tuples for each input Tuple in the dominant actuating stream and not for the reference data stream.

Some sample code that illustrates how that implementation might be supported is shown in Example 3-8.

Example 3-8 Streams Application code of the Consolidation pattern stereotype

```
[Application]
  Consolidation
[Program]

vstream Schema1 (sourceID: String, sourceID2: Integer, value1: Double)

vstream Schema2 (sourceID: String, value2: Double)

vstream Schema3 (sourceID2: Integer, value3: Integer)

stream RawValues (schemafor(Schema1))
  := Source ()
    ["file:///input1.dat", csvformat, nodelays, initDelay=5]
    {}

stream ConsolidatingValues (schemafor(Schema2))
  := Source ()
    ["file:///input2.dat", csvformat, nodelays]
    {}

stream ConsolidatingValues2 (schemafor(Schema3))
  := Source ()
    ["file:///input3.dat", csvformat, nodelays]
    {}

stream Consolidation1 (sourceID: String, sourceID2: Integer, value1:
Double, value2: Double)
  := Join (RawValues<count(1),perGroup> ;
    ConsolidatingValues<count(1),perGroup> )
    [ {sourceID} = {sourceID} ]
    {$1.sourceID, sourceID2, value1, value2}
```

```

stream Consolidation2 (sourceID: String, sourceID2: Integer, value1:
Double, value2: Double, value3: Integer)
  := Join (Consolidation1<count(1),perGroup> ;
          ConsolidatingValues2<count(1),perGroup> )
  [ {sourceID2} = {sourceID2} ]
  {$1.sourceID, $1.sourceID2, value1, value2, value3}

Nil := Sink (Consolidation2)
      ["file:///output.dat", csvformat, nodelays]
      {}

```

Suggestions

The join Operators need to keep windows of data for an appropriate period of time to allow the matching of Attributes between streams. To keep the window sizes to a minimum, incoming streams should be filtered to remove unnecessary Tuples before being processing in a join.

3.2.9 Merge pattern: Merging multiple streams

When you are consuming streams from multiple sources, it is possible that the formats of the data from each source are different, even if the information content for each stream is the same.

In general, you want to process the data from the streams in a common way so it is desirable to convert these streams into a standard format and merge them together before being consumed by the rest of the Streams Application.

Example

In the example of child location service (see 2.3, “End-to-end example: Streams and the lost child” on page 49), you have multiple sources of data, all of which inform us about the location of a child at a particular time. The sources are room door scans, retail purchases, and access to the ski lift lines. From each source, you can infer the location of the child in question. You want to standardize and merge the streams and then process them as a single logical stream. A simple implementation of the Merge design pattern is shown in Figure 3-19.

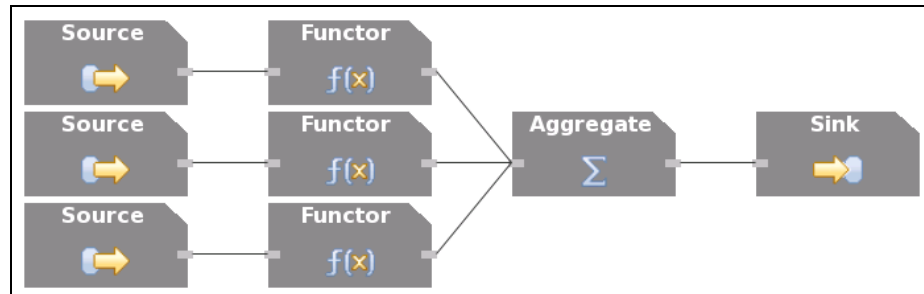


Figure 3-19 Application topology of the Merge pattern stereotype

Stereotype

In this illustration, separate Source Operators read in the different data streams. For each stream, a Functor Operator converts the varying formats into a standard shared schema before they are consumed by the rest of the application, in this case an aggregate Operator.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. Three Source Operators each read in a data stream from an input file, creating the Source1, Source2, and Source3 streams. The schemas of the streams are different, although they all contain f2 and f4 Attributes.
2. Each Source stream is processed by a different Functor Operator, producing a stream in the standard format required for the next Operator (containing only the f2 and f4 values), which is the aggregate.
3. The aggregate Operator takes the standardized streams, merges the contents, and deals with the contents as a single logical stream. The aggregate generates an average value of the data from all three sources together.
4. A Sink Operator takes the Results stream and outputs the data to a file named `output.dat`, which will be used by the relevant recipients.

Variations

In simple cases of Source processing, the format conversion may be performed by customizing the Source Operator itself. However, in this case, transforming Functors are not required.

Note: The stereotype shown in Example 3-9 is simple enough not to require transforming Functors, that is, the selection of the values could be performed in the Source Operators. The Functors are included to illustrate the approach that would be taken should more complex transformation be required.

Example 3-9 Streams Application code of the Merge pattern stereotype

```
[Application]
  Merge
[Program]

vstream vSource1 (f1 : Integer, f2 : String, f3 : Double, f4 : Float,
f5 : String)
vstream vSource2 (f2 : String, f3 : Double, f4 : Float)
vstream vSource3 (f1 : Integer, f2 : String, f4 : Float, f5 : String)
vstream StandardFormat (f2 : String, f4 : Float)

stream Source1 (schemaFor(vSource1))
  := Source ()
    ["file:///input1.dat", csvformat, nodelays]
    {}

stream Source2 (schemaFor(vSource2))
  := Source ()
    ["file:///input2.dat", csvformat, nodelays]
    {}

stream Source3 (schemaFor(vSource3))
  := Source ()
    ["file:///input3.dat", csvformat, nodelays]
    {}

stream Standardised1 (schemaFor(StandardFormat))
  := Functor (Source1) [] {}

stream Standardised2 (schemaFor(StandardFormat))
  := Functor (Source2) [] {}

stream Standardised3 (schemaFor(StandardFormat))
```

```

:= Functor (Source3) [] {}

stream Results (f2 : String, avgf4 : Float)
  := Aggregate (Standardised1, Standardised2, Standardised3<count(10),
count(5), perGroup> )
    [f2]
    {Any(f2), Avg(f4)}

Null := Sink (Results)
      ["file:///output.dat", csvformat, nodeLays]
      {}

```

Suggestions

This pattern should be used as appropriate.

3.2.10 Integration pattern: Using existing analytics

Integration of existing analytic packages/algorithms into Streams allows the value to be enhanced.

The analysis of stream data can rely on complex algorithms, which might have already been developed and deployed. It is undesirable to have to replicate this logic, and is less expensive and more convenient to integrate in its existing form.

This situation can be supported in Streams by using a Functor, UDOP, or UBOP to integrate the existing logic.

Example

An example of this pattern is the language translation of web logs (blogs). A multi-national company performing market research may analyze the blogs of consumers on the web, regardless of the language of the text. Sophisticated systems exist to translate text from one language to another. A package can be integrated with Streams to allow the analysis of multi-lingual sources. A simple implementation of the Integration design pattern is shown in Figure 3-20.



Figure 3-20 Application topology of the Integration pattern stereotype

Stereotype

In this illustration, a Source Operator reads in the data stream, then you introduce a user-defined Operator (UDOP) that is coded to call to an external algorithm. The result of the external algorithm is put into the UDOP output stream. The Sink Operator passes the results to the relevant recipients.

Stereotype walkthrough

The following are brief descriptions of the steps involved in the stereotype:

1. A Source Operator reads in the data stream from the `input.dat` file, creating the `RawValues` stream.
2. A user-defined Operator (UDOP) processes the `RawValues` stream, calls the external algorithm, and incorporates the output `value2` into the `Results` stream.
3. A Sink Operator takes the `Results` stream and outputs the data to a file named `output.dat`, which will be used by the relevant recipients.

Variations

A range of variations for including the external logic are discussed in 3.3, “Using existing analytics” on page 120.

Some sample code that illustrates the integration pattern of the stereotype is shown in Example 3-10.

Example 3-10 Streams Application code of the Integration pattern stereotype

```
[Application]
  Integration
[Program]
```

```
vstream Schema1 (sourceID: String, value1: Double)
```

```
stream RawValues (schemafor(Schema1))
  := Source ()
    ["file:///input1.dat", csvformat, nodelays]
    {}
```

```
stream Results (sourceID: String, value1: Double, value2: Double)
  := Udob (RawValues)
    [ "ExternalCall" ] {}
```

```
Nil := Sink (Results)
    ["file:///output.dat", csvformat, nodelays]
    {}
```

Suggestions

This pattern should be used as appropriate.

3.3 Using existing analytics

The standard Operators in Streams provide a useful and wide ranging toolkit for assembling applications to deliver stream computing systems. However, you may find that you have existing analysis systems that you want to use to process your data sources. You want the Streams system to use the same analytics, either to take advantage of the Streams scheduler to deliver high volumes and scalability, or to allow the combination of multiple sources and different analyses.

It may be possible to reimplement existing functionality in standard streams Operators, but the more complex this functionality is, the more likely you are to want to reuse the current systems and avoid the development cost of reimplementing the intricate logic.

As a simple example, suppose that you are receiving streams of data from all over the world and each Tuple you get includes a time stamp, but the time stamps that you receive are all from different time zones from different continents. To be able to compare times, you should convert them all to a common time zone, for example, Greenwich Mean Time (GMT). You need to introduce logic into Streams that will take a time, a date, and a time zone label and convert the time into GMT. This logic is reasonably complicated, as the time zones vary depending on the exact date, as different areas have different daylight saving schemes that move the clocks back and forward an hour. Fortunately, this is code that already exists, so you can include this logic into Streams without the need to write it yourself.

Streams allows you to incorporate existing analytics software in a number of different ways. In this section, we introduce the possibilities and illustrate them with an example:

- *User-defined function (UDF)*: Streams Functor Operators can invoke C++ functions to process Attributes of the data streams. There is a large set of built-in functions for operating on different data types, lists, and so on. You can include your own C++ code as a user-defined function, or a package of user-defined functions that can be invoked from a Streams Functor Operator.

Figure 3-21 on page 121 show a simple flow depicting how you can use a UDF to convert time stamps between time zones. You include a Functor Operator in the stream, which takes the RawInput, including the time stamp, time zone, and date. The Functor allows you to declare a state variable (`$gmt`), include a C++ package reference (`MyTimeConversion.h`), and then to

call a function from this package (convertToGMT). The resulting value is assigned to the GMTTime Attribute.



Figure 3-21 Application topology example for UDF extension

Example 3-11 shows some sample code for implementing a UDF extension.

Example 3-11 Streams Application code for UDF extension

```
[Application]
    GMTConversionExamples
[Libdefs]
package "MyTimeConversion.h"
[Program]
...
stream StandardTimeInput (schemafor(RawInput1), GMTTime : String)
    := Functor (RawInput1)
    < # functor state declaration
        String $gmt := "";
    > # end of functor state declaration
    < # functor custom code
        $gmt := convertToGMT (TimeStamp, TimeZone, Date);
    > # end of functor custom code
        []
        {GMTTime := $gmt}
...

```

- *User-defined Operator (UDOP)*: Streams provides the possibility of including your own logic within a user-defined Operator.

The UDOP allows you to include external code libraries and process Tuples from the input streams of the Operator, one at a time, submitting output Tuples as appropriate. UDOPs can be codes in C++ or Java (with some limitations).

Figure 3-22 shows a flow about how you use a UDOP to convert time stamps between time zones. You include a UDOP Operator in the stream, which takes the RawInput, including the time stamp, time zone, and date. The UDOP Operator is named *convertToGMT*. The code for this is expected to be present in the application's compilation directories. The UDOP code will process the input streams, expecting the TimeStamp, TimeZone, and Date Attributes to be present. The UDOP will generate and submit an output Tuple for each input Tuple received, calculate the GMT time, and add this data to the output stream.

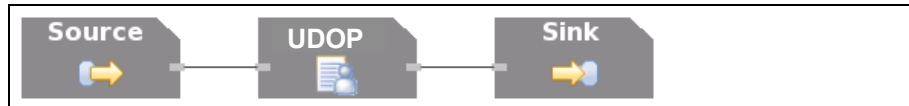


Figure 3-22 Application topology example for UDOP extension

Example 3-12 shows some sample code for implementing Streams Application code for a UDOP extension.

Example 3-12 Streams Application code for UDOP extension

```
[Application]
  GMTConversionExamples
[Program]
...
stream StandardizedTime (schemafor(RawInput), GMTTime : String)
  := Udup (RawInput)
    ["convertToGMT"]
    {}
...
```

- *User-defined built-in Operator (UBOP)*: UBOPs provide similar capabilities as UDOPs, but appear to the Streams system as though they are built-in Operators. It is possible to include external code libraries into a UBOP to integrate with existing code.

UBOPs should flexibly support different input and output streams formats, allowing the Operator to be reused in many different situations. This is in contrast to a UDOP, which is likely to be limited to specific input and output streams.

Figure 3-23 shows how you can use a UBOP to convert time stamps between time zones. You have a *use clause* to tell the application where to find the definition of the UBOP (in this case, you reference the `convertToGMT` UBOP within the `com.ibm.streams.timeconversion` toolkit). The Streams compiler will expect this Operator to be included in the Streams installation directories.

The UBOP Operator takes the `RawInput`, including the time stamp, time zone, and date. The UBOP code will process the input streams, expecting the `TimeStamp`, `TimeZone`, and `Date` Attributes to be present. The UBOP will generate and submit an output Tuple for each input Tuple received, calculate the GMT time, and add this to the output stream.



Figure 3-23 Application topology example for UBOP extension

Example 3-13 shows sample application code for a UBOP extension.

Example 3-13 Streams Application code for UBOP extension

```
[Application]
    GMTConversionExamples
[Program]
...
use com.ibm.streams.timeconversion.convertToGMT
stream StandardizedTime (schemafor(RawInput), GMTTime : String)
    := convertToGMT (RawInput)
    []
    {}
...

```

- Java UDOP, C++ UDOP, or UBOPs can also be used to perform a call to a software process executing outside the Streams run time. The UDOP or UBOP performs the role of an adapter to a specific Application Programming Interface (API) or a specific communications protocol.

Note: Calls out to external systems may take a significant amount of time due to issues such as communication latency. Waiting for the external system to respond to each Tuple could cause the system to run too slow.

For example, suppose you have a requirement to process 10,000 Tuples a second. If a call-out takes 1 millisecond per call, then if you process these Tuples one at a time, the most you could process would be 1,000 Tuples per second.

To overcome this problem, you need to process Tuples in parallel. In Example 3-13, you need to process at least 10 Tuples at the same time to achieve a throughput of 10,000 Tuples a second. However, consider the following items:

- ▶ It is possible to create a multi-threaded UDOP and process Tuples in separate threads.
 - ▶ Another alternative would be to segment the stream into a number of substreams and process the Tuples in separate UDOPs, parallelized by the segmented stream.
-
- ▶ You can use the existing analytic package to pre-process the data streams and perform their processing before the stream of data reaches the Streams run time.

Figure 3-24 shows a data source being preprocessed by an analytics package before it is consumed by the Streams Application. One downside to this approach is that the external package is not able to take advantage of the Streams run time to schedule workload and to automatically handle stream input and output.

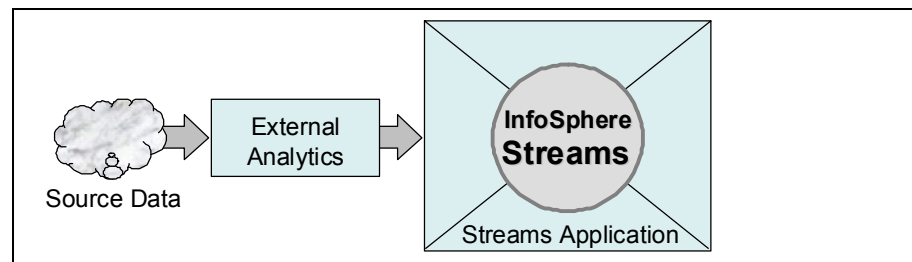


Figure 3-24 Pre-processing data sources through external analytics software

- ▶ You can use the existing analytics package to post-process the data streams, perform the processing after the data has been through the Streams Application.

Figure 3-25 shows a data source being post-processed by an analytics package after it has been processed by the Streams Application. One downside to this approach is that the external package is not able to take advantage of the Streams run time to schedule workload and to automatically handle stream input and output.

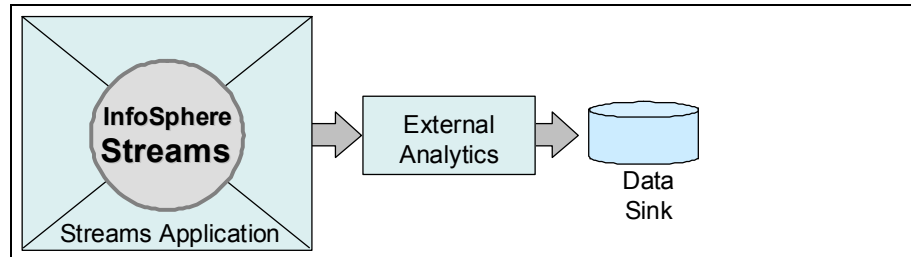


Figure 3-25 Post-processing of data sources using external analytics software



Deploying IBM InfoSphere Streams Applications

In this chapter, we take a closer look at the runtime environment of the IBM InfoSphere Streams (Streams) system and how to deploy Streams Applications to this environment. The following is a list of topics included in this chapter:

- ▶ A description of the runtime architecture, specifying the services that form the Streams runtime components and how they interact.
- ▶ An explanation of how the Streams services are deployed into different topologies, particularly single Host and multiple Host Instances.
- ▶ A discussion of the approaches to size suitable hardware for a Streams system.
- ▶ A discussion of the application deployment strategy and how an application is segmented into Processing Elements (PE), which are then deployed to the runtime environment.
- ▶ A description of the impact of the runtime topology on the design and structure of the Streams Application. It may be necessary to segment the application to allow flexible deployment or to overcome performance bottlenecks.
- ▶ A discussion of the aspects of performance engineering that outlines an approach to testing and optimizing performance to meet performance requirements.

- ▶ A discussion of the aspects of Streams security, including secure communication between streams Hosts, access control, and how to control the PE functionality.
- ▶ A discussion of the built-in failover recovery features of Streams, including application element recovery and recovery of streams services.
- ▶ A discussion of the performance counter interface built into Streams and how it can be used to monitor a running Streams system.

4.1 Runtime architecture

The Streams Runtime consists of a number of service processes that interact to manage and execute the Streams Applications. A complete set of these services act together and form a *Streams Instance*. Separate Instances can be created and configured independently from each other, even though they may share some of the same physical hardware. Figure 4-1 shows a Streams Instance with its constituent services.

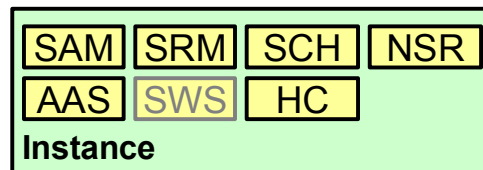


Figure 4-1 Services associated with the Streams Instance

The services that comprise an Instance are:

- ▶ Streams Application Manager (SAM)
SAM is a management service that administers the applications in the Streams run time. Specifically, SAM handles job management tasks, including user requests, such as job submission and cancellation. It interacts with the Scheduler to compute placement for the Processing Elements that comprise an application. It also interacts with the Host Controller to deploy and cancel the execution of these Processing Elements.
- ▶ Streams Resource Manager (SRM)
This is a management service that initializes the Streams Instance, and monitors all its services. The SRM collects and aggregates system-wide metrics, including the health of Hosts that are part of an Instance and the health of the Streams components, as well as relevant performance metrics necessary for scheduling and system administration by interacting with the Host Controller.

- ▶ Scheduler (SCH)

The Scheduler is a management service that computes placement decisions for applications to be deployed on the Streams runtime system. It interacts primarily with the SAM service to handle job deployment requests. It also interacts with the SRM component to obtain runtime metrics necessary for computing Processing Element placement decisions.

- ▶ Name service (NSR)

This is a management service that stores service references for all the Instance components. This service is used by all Instance components to locate the distributed services in the Instance for purposes of inter-component communication.

- ▶ Authorization and Authentication Service (AAS)

AAS is a management service that authenticates and authorizes operations for the Instance.

- ▶ Streams Web Service (SWS)

This management service provides web-based access to the Instance's services. This service is optional and can be added or removed from the Instance as required.

- ▶ Host Controller (HC)

The HC application service runs on every application Host in an Instance. This service carries out all job management requests made by the SAM service, which includes starting, stopping, and monitoring Processing Elements.

- ▶ Processing Element Container (PEC)

Each running Processing Element (PE) is hosted in a PEC process that is started and monitored by the Host controller for the Host where the PE is running.

The PEC services are run when applications are started in the Streams Instance and are not shown in the runtime diagrams.

4.2 Introduction to topologies

In this section, we discuss Streams topologies, describing how to create Instances with single and multiple Hosts, and how to start and stop these Streams Instances.

4.2.1 Single Host Instance

In the simplest case, a single Host computer is all that is required to perform the Streams processing. This is a configuration that is suitable for lower volume applications, or developing and testing applications. It is shown in Figure 4-2.

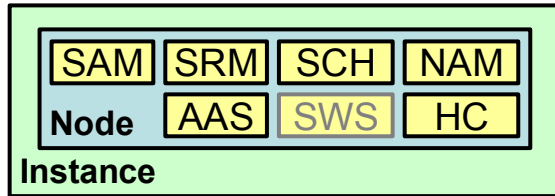


Figure 4-2 Single Host Streams run time

Creating a single Host Instance

To create a single Host Instance, you can issue one of the commands shown in Example 4-1 from an account that is configured for the Streams environment.

Example 4-1 Making a Streams Instance with a single Host

```
$ streamtool mkinstance -i newInstance --hosts RBCompute1
or
$ streamtool mkinstance -i newInstance --numhosts 1
or
$ streamtool mkinstance -i newInstance --hfile ~/hostfile
where
$ cat ~/hostfile
RBCompute1
```

Each alternative command creates an Instance with the name *newInstance*:

- ▶ The `--hosts RBCompute1` option creates an Instance with a Host named “RBCompute1”.
- ▶ The `--numhosts 1` option creates an Instance with a single Host. The Streams Resource Manager (SRM) service will choose an available Host from its set of known hosts.
- ▶ The `--hfile ~/hostfile` option allows you to include the path name to a file which contains a list of hosts. In this case `~/hostfile` will be a file that contains a single Host name.

You can check the creation of the Instance by using the commands shown in Example 4-2.

Example 4-2 Checking the configuration of a Streams Instance with a single Host

```
$ streamtool lsinstance
newInstance@streamsadmin
$ streamtool lshosts -i newInstance -long
Host                               Services
RBCompute1                        hc,aas,nsr,sam,sch,srm,sws
```

The output from the **lsinstance** command shows that there is a single Instance called *newInstance@streamsadmin*. Note that streamsadmin is your login name, which is appended to the requested Instance name.

The output from the **lshosts** command shows that *newInstance* has a single Host called RBCompute1, which has a list of associated services, that is, hc, aas, nsr, sam, sch, srm, sws. See 4.1, “Runtime architecture” on page 128 for the definition of these services.

Optional SWS service

Note that in Figure 4-2 on page 130 that the SWS service is gray. This is because it is an optional service that provides a web administration capability that can be added or removed from the Instance as required.

Example 4-3 shows three commands for managing the SWS service. They are used to:

1. Create an Instance with no sws service.
2. Add the sws service to an Instance.
3. Remove the sws service from an Instance.

Example 4-3 Adding and removing the optional SWS service

```
$ streamtool mkinstance -i newInstance --numhosts 1 --rmservice sws

$ streamtool addservice -i newInstance --host RBCompute1 sws

$ streamtool rmservice -i newInstance --host RBCompute1 sws
```

4.2.2 Multiple Hosts Instance

When multiple hosts are needed to deliver the necessary throughput and latency, an Instance can be defined with multiple hosts. The services of the Instance are distributed between those hosts, as shown in Figure 4-3.

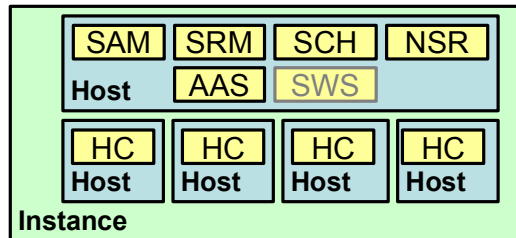


Figure 4-3 Multiple Host streams run time

Creating a multiple Host Instance

To create a multiple Host Instance, you can issue one of the commands shown in Example 4-4 from an account that is configured for the Streams environment.

Example 4-4 Making a Streams Instance with multiple hosts

```
$ streamtool mkinstance -i newInstance --hosts RBManage, RBCompute1,
RBCompute2, RBCompute3, RBCompute4
or
$ streamtool mkinstance -i newInstance --numhosts 5
or
$ streamtool mkinstance -i newInstance --hfile ~/hostfile
where
$ cat ~/hostfile
RBManage
RBCompute1
RBCompute2
RBCompute3
RBCompute4
```

Each command in Example 4-4 creates an Instance with the name newInstance, with the following options:

- ▶ The --hosts RBManage, RBCompute1, RBCompute2, RBCompute3, RBCompute4 option creates an Instance with hosts named RBManage, RBCompute1, RBCompute2, RBCompute3, and RBCompute4.

- ▶ The `--numhosts 5` option creates an Instance with five hosts. The Streams Resource Manager (SRM) service will choose available hosts from its set of known hosts.
- ▶ The `--hfile ~/hostfile` option allows us to include the path name in a file that contains a list of hosts, in this case, `~/hostfile` contains a list of the five hosts for the Instance.

You can check the creation of the Instance, as shown in Example 4-5.

Example 4-5 Checking the configuration of a Streams Instance with multiple hosts

```
$ streamtool lsinstance
newInstance@streamsadmin
$ streamtool lshosts -i newInstance -long
Host                               Services
RBManage                           aas,nsr,sam,sch,srm,sws
RBCompute1                         hc
RBCompute2                         hc
RBCompute3                         hc
RBCompute4                         hc
```

The output from the `lsinstance` command shows that there is a single Instance called `newInstance@streamsadmin`. Note that `streamsadmin` is our user login name. That user name is appended to the requested Instance name.

The output from the `lshosts` command shows that `newInstance` has five hosts named `RBManage`, `RBCompute1`, `RBCompute2`, `RBCompute3`, and `RBCompute4`:

- ▶ `RBManage` has a list of associated services, which are `aas`, `nsr`, `sam`, `sch`, `srm`, and `sws`. `RBManage` is a management Host because it runs the management services of the Instance.
- ▶ The `RBCompute` hosts each have a single associated service, which is `hc`, the Host Controller. `RBCompute` is an application Host. When application jobs are run in the Instance, the Host Controller service will run and control Processing Element Controllers on each of these hosts.

Just as for the single Host Instance, the Streams Web Service (SWS) is optional. Example 4-3 on page 131 shows how to add and remove the SWS from an Instance.

Reserved management hosts

In Example 4-5, you can see that a single Host has been assigned all the management services, and it has been reserved so that it does not run the Host Controller service and so will not run Processing Elements.

If the management Host is being overloaded, then it is possible to use the **streamtool mkinstance** command with the **--reserved <n>** option to reserve more than one Host for management services, and to reallocate services between those management hosts.

Dynamic Host selection

When you make an Instance by using the **--numhosts** option, Streams refers to a hosts file that defines the set of known hosts for this installation of Streams. The default hosts file is `<streamsadmin home>/.streams/config/hostfile`.

A Streams installation host file example is shown in Example 4-6.

Example 4-6 Streams installation host file

```
$ cat ~/.streams/config/hostfile
RBManage, build, execution
RBCompute1, execution
RBCompute2, execution
RBCompute3, execution
RBCompute4, execution
RBCompute5, execution
RBCompute6, execution
```

You can see that each Host is specified on a separate line, and the hosts can be defined as a build Host, execution Host, or both. In Example 4-6, all build activity will take place on the RBManage node, and RBCompute nodes will be used to execute applications.

4.2.3 Starting and stopping the instance

After creating a Streams Instance, the Instance needs to be started before it can run Streams Applications. Starting an Instance causes the services to be started on the appropriate hosts.

Example 4-7 shows the use of the **streamtool startinstance** command to start the inst Instance.

Example 4-7 Starting a Streams Instance

```
$ streamtool startinstance -i inst
CDISC0059I Starting up Streams instance 'inst@streamsadmin'...
CDISC0078I Total instance hosts: 1
CDISC0056I Starting a private Distributed Name Server (1 partitions, 1
replications) on host 'redbookstreams'...
```

```
CDISC0057I The Streams instance's name service URL is set to
DN:redbookstreams:7389.
CDISC0061I Starting the runtime on 1 management hosts in parallel...
CDISC0003I The Streams instance 'inst@streamsadmin' was started.
```

You can check the status of the Instance with the **streamtool lsinstance** and **streamtool getstatus** commands, as shown in Example 4-8.

Example 4-8 Showing the status of a started Streams Instance

```
$ streamtool lsinstance
inst@streamsadmin

$ streamtool getstatus -i inst
Instance: inst@streamsadmin Started: yes State: READY Hosts: 1 (1/1
Management, 1/1 Application)
Host          Status  Schedulable  Services
redbookstreams READY    yes          READY:aas,hc,nsr,sam,sch,srm,sws
```

Stopping a Streams Instance stops the services running on their hosts, which frees up the resources used to run the Streams Instance. Example 4-7 on page 134 shows the use of the **streamtool stopinstance** command to stop the inst Instance.

Example 4-9 Stopping a Streams Instance

```
$ streamtool stopinstance -i inst
CDISC0063I Stopping instance 'inst@streamsadmin'...
CDISC0050I Stopping service 'hc' on host 'redbookstreams'...
CDISC0050I Stopping service 'sws' on host 'redbookstreams'...
CDISC0050I Stopping service 'sam' on host 'redbookstreams'...
CDISC0050I Stopping service 'sch' on host 'redbookstreams'...
CDISC0050I Stopping service 'srm' on host 'redbookstreams'...
CDISC0050I Stopping service 'aas' on host 'redbookstreams'...
CDISC0068I Cleaning up the runtime services on 1 (all) hosts in
parallel...
CDISC0054I Stopping and cleaning up the private Distributed Name
Service on 1 hosts in parallel:
redbookstreams
CDISC0055I Resetting the Streams instance's name service location
property.
CDISC0004I The Streams instance 'inst@streamsadmin' was stopped.
```

4.3 Sizing the environment

Part of designing a Streams system is sizing the hardware to determine the type and quantity of computers needed to support the envisaged applications with the necessary throughput and speed.

Sizing the hardware for an currently undeveloped application is a difficult activity. You need to determine the amount of hardware required to execute the application, and which steps of the application are the most processor and network intensive. To do accomplish this task, you need to consider the following criteria:

- ▶ Volumes of data flowing through the data sources, which includes average volumes and peak volumes.
- ▶ Estimated data reduction at various stages of processing. Wherever possible, you should reduce the data volumes by filtering so that you only process data items that are of interest.
- ▶ Complexity of the Streams Operators. It is useful to assess, as best as you can, the processing requirement of the Operators and flows within streams. This task can be accomplished by prototyping and measuring the workload and resource usage of the prototype or by comparing the prototype with Streams Applications that have similar complexity.
- ▶ Model the total workload. You should scale the measured or estimated processing requirements by the projected workload volumes to produce a projection of the processing requirement, that is, how many compute hosts are required to support the envisaged workload.

There is a trade-off as to whether you size a system to cope with average load, peak load, or somewhere in between:

- ▶ Sizing to the peak volumes means that you can achieve your throughput and latency targets more frequently, but the machines will be underutilized for periods of time.
- ▶ Sizing to the average throughput rate will result in needing less hardware than for a peak volumes sizing, which results in a lower cost for the overall system. However, the latency of end-to-end Tuple processing is likely to increase during times of above-average traffic. This will, of course, be undesirable in low latency focused systems.

4.4 Deployment implications for application design

Having determined the size of the run time required for the projected applications with their required volumes, you can determine how the processing should be divided across multiple hosts. Consider the following items:

- ▶ Can you execute your Streams Application on a single Host Instance?
- ▶ Can the processing load predicted for each individual Operator be satisfied by a single Host?

The Streams Scheduler allocates Processing Elements (groups of Operators) to act as whole units to hosts and is not able to segment processing intensive Operators.

- ▶ If any Operator cannot satisfy the required volumes on a Host, then you need to distribute the Operator workload. Options for this are outlined in the following sections and include considerations for:
 - Running multiple identical Operators in parallel
 - Segmenting the Operator functionality into multiple pipelined stages
 - Filtering or aggregating the stream to reduce the load on the Operator

4.4.1 Dynamic application composition

The Streams run time allows you to run multiple applications in a Streams Instance, and to export a stream from one application and import the stream into other applications. This is shown in Figure 4-4, where the Sensor Adapter application (at the top left) exports a stream of cleansed, filtered Tuples to the Sensor Detailed Analysis application (illustrated by the arrow pointing to that application) at the bottom right.

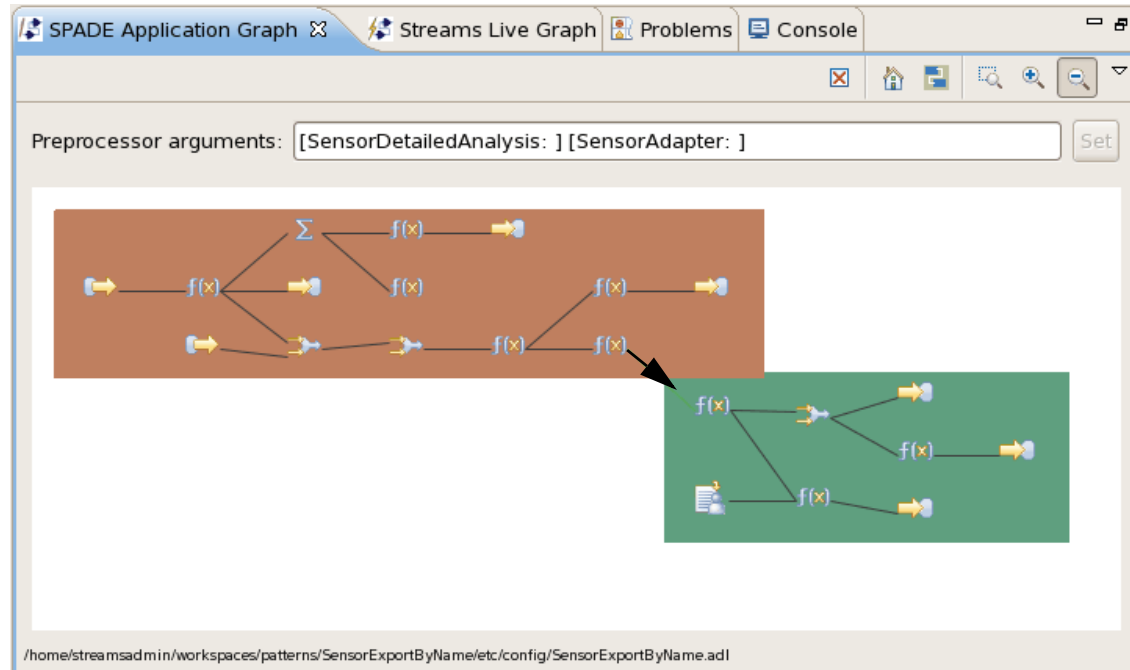


Figure 4-4 Application composition by exporting and importing streams

The exporting and importing of streams between applications allows you to use applications together to form larger composite applications. However, you are able to start and stop the individual applications independently without affecting the other applications.

For example, you could choose to segment the Streams Applications into three different categories, such as:

- ▶ Applications to read, validate, and filter individual Source streams
- ▶ Applications to analyze a number of the Source streams
- ▶ Applications to format and deliver output data to Sinks

This segmentation allows you to stop and start the Source Applications when the stream availability varies. You can run multiple analysis applications, and develop and deploy new analysis algorithms, without impacting the existing ones. You can also independently start and stop the Sink applications to support new downstream applications and to relieve load on the system when downstream applications are not active.

When exporting a stream from an application, you declare the stream with the *export* keyword. When importing a stream into a new application, you declare the stream with the *import* keyword and then define which streams this import is *tapping*, that is, which streams will be used for the importing.

Exported streams and imported streams can be matched in two different ways:

- Export and import by name. You can export a stream, and import it using the specific application and stream name, as shown in Example 4-10. This is useful when you only want to import a single, known Instance of a stream from a single, known application.

Example 4-10 Example of importing and exporting streams by name

[Application]

SensorAdapter

[Program]

Export by name

export

```
stream SensorReadings(sensor:String, reading:Integer)
  := Functor(RawSensorData)
    [...]{...}
```

[Application]

SensorDetailedAnalysis

[Program]

Import specific known stream

import

```
stream IndividualSensorReadings(sensor:String, reading:Integer)
  tapping SensorAdapter.SensorReadings
```

- Export and import by properties. You can export a stream and specify a number of properties for that stream. When importing, you can specify the properties of the streams that you want to tap and this will dynamically identify matching exports in the Streams Instance. This approach is shown in Example 4-11.

Example 4-11 Example of importing and exporting streams by properties

[Application]
SensorAdapter

```
[Program]
# Export by stream properties
export
properties [ category: "sensor";
              id : "1"]
stream Sensor1Readings(reading:Integer)
:= Functor(SensorReadings)
   [sensor="1"]{}
```

[Application]
SensorDetailedAnalysis

```
[Program]
# Import streams by matching properties:
import
stream MergedSensorReadings(reading:Integer)
   tapping "stream[category='sensor']"
```

Note: If you export a stream from one application and it is not imported elsewhere, then the Tuples in this stream are lost, that is, they are not automatically persisted or buffered.

When you start an importing application, the contents of the stream are tapped from that point in time onwards. Likewise, if an importing application is stopped for a period of time and restarted, then the contents of the stream will be lost during the time the application is stopped.

4.4.2 Overcoming performance bottleneck Operators

As outlined in 3.2.3, “Parallel pattern: High volume data transformation” on page 96, when you deploy your application on the runtime hardware, you may find that you have a single application Operator that is a performance bottleneck.

This means that the Operator cannot deliver the necessary throughput or response time to satisfy the system performance requirements.

If you leave this Operator unchanged, it will throttle the rate of throughput, and require you to spread the workload over multiple hosts.

One option to spread the workload is to segment the stream into a number of substreams, each of which can be processed in parallel on different Host computers. This is shown in the Parallel pattern in Figure 4-5.

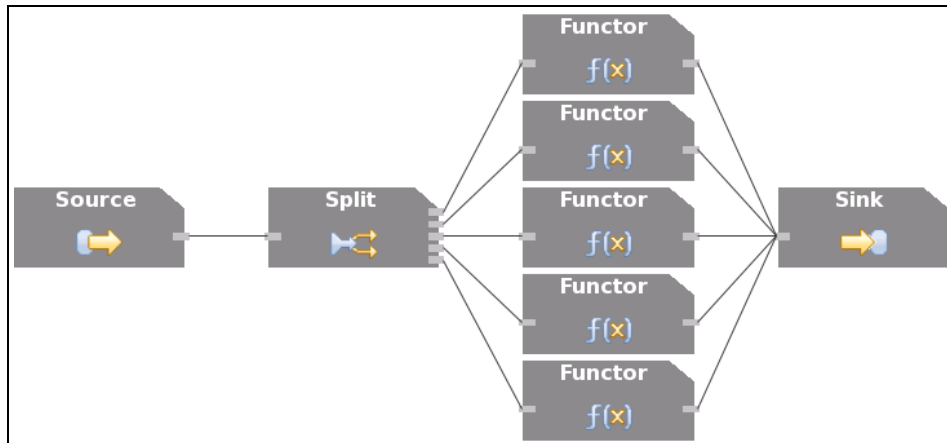


Figure 4-5 Parallel pattern

The problem of state

Stream computing works on the general principle that applications are stateless and that data passes through the system and is not stored. However, a number of the Streams Operators are not totally stateless, and they can be configured to keep state between subsequent Tuples. When this is the case, splitting a stream for parallel processing can change the results of the Operators.

The following list outlines the state management possibilities for Streams Operators:

- ▶ Stateless Operators:
 - Source
 - Sink
 - Barrier
 - Split

- ▶ Potentially stateful Operators:
 - Functor: Generally, a Functor is stateless unless it is customized to include a Functor state declaration portion. It is also possible to refer to historical Tuples that are then buffered in the Functor Operator.
 - Puncctor: Can use historical Tuples from the stream (buffered by the Operator) to determine when to punctuate a stream.
 - User-defined Operators (UDOPs): Can implement any level of state persistence and may invoke external systems that also could maintain a state history.
 - User-defined Built-in Operators (UBOPs): Like UDOPs, they can implement any level of state persistence and may invoke external systems that can maintain state history.
- ▶ Stateful Operators:
 - Join: You can define windows on the input streams to allow the join to operate on a recent portion of each stream. You can also define a perGroup modifier to split the window by a particular Attribute's values, in which case a window of Tuples is held for each different value of the Attribute.
 - Aggregate: Windows® can be defined on the input streams to allow the aggregation of a recent portion of the stream. The window may be segmented by defining a perGroup modifier.
 - Sort: Windows can be defined on the input streams. The Tuples from the input stream are bunched and sorted according to the defined windows.
 - Delay: Delays streams by storing a buffered list of Tuples. When the right time arrives, the Tuples are released into the outgoing stream.

You may want to segment a stream to distribute its workload, but be aware that this can change the behavior of the stateful Operators, as they will now only receive a subset of the original stream.

For example, consider a Join Operator, which joins data from multiple high volume streams. If these streams are segmented to distribute the processing load, the data items that would have matched in a single Operator will be spread out over different Join Operators and the number of matches could be reduced.

Figure 4-6 shows an illustration of segmenting a Join Operator:

- ▶ In the single join case, the Operator processes three Tuples (1,2,3) from both streams. Three matches will be output from the Join Operator.
- ▶ In the split join case, the streams have been segmented and processed by three separate Join Operators. Only one of the Tuple pairs will match in this case.
- ▶ In the partitioned join case, we have ensured that the streams are segmented to send related Tuples to the same Join Operator. All three matches will be identified and output in this case.

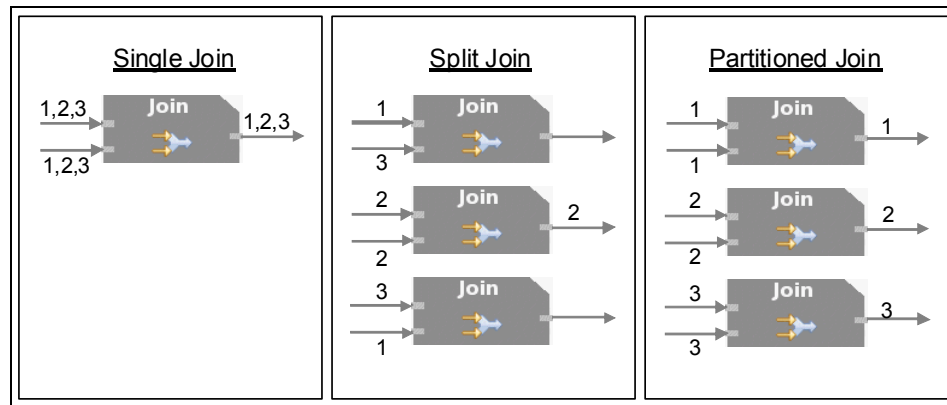


Figure 4-6 Side effects of splitting a Join Operator

A stream that is processed by an Aggregate or Join Operator with a perGroup clause can be segmented by that perGroup Attribute with no change in processing behavior. For example:

- ▶ Consider the example of a stock trading application where you calculate the average price of a stock for each company in your portfolio. You can split the stock price stream by stock symbol, as Tuples will be kept together and your aggregate will be unaffected.
- ▶ In contrast, if you want to keep an average of all stock quotes together, segmenting the streams by stock symbol will change the aggregate calculation.

Pipelining as an alternative

An alternative to splitting a stream and processing in parallel is to use the Pipeline pattern (Section 3.2.4, “Pipeline pattern: High volume data transformation” on page 101). Using this pattern, processing is divided into multiple stages that can be performed one after another.

Note that pipelining has a larger communication impact than parallelism. There is a processing impact and a time penalty in communicating between Processing Elements, especially when the PEs are on different hosts.

In the case of pipelining, every Tuple is sent to every Operator, so if you have five separate Operators pipelined together, and each Operator is executing on a separate hosts, you have to send the Tuples across all five hosts.

In contrast, if you segment a stream into five parallel Operators, then each Tuple only needs to be sent between hosts twice (once to the parallel Operator and then back again), but you incur a processing impact by segmenting the streams.

4.5 Application deployment strategy

Application deployment addresses the issue of how the components of the application are distributed and executed on the physical hosts. There are a number of ways that the application may be segmented and grouped to optimize throughput, latency, and scalability:

At development time:

- ▶ An application may be segmented into multiple applications, and then Streams may be exported from one application and imported into another.
- ▶ You may choose to deploy multiple copies of an Operator, each working on a subset of the total stream. This allows parallel processing when the Operator is processor intensive and cannot satisfy the throughput requirements on a single Host.
- ▶ You may choose to segment processor intensive analysis into multiple stages and process the stream one stage after the other. This allows pipelined processing, and segmenting the processing load of the Operators.

At compile time:

The compiler combines Operators into Processing Elements (PEs). These are the individual execution elements that are deployed to the run time. The compiler can intelligently optimize execution by determining which Operators should be combined together and which should be segmented.

At run time:

- ▶ The Streams Application Manager (SAM) accepts requests to run applications. SAM invokes the Scheduler service to determine on which hosts the various PEs should run. The Scheduler then retrieves Host information

and performance metrics from the Streams Resource Manager which enables the best placement of PEs to be determined.

- The PEs are deployed to the hosts as determined by the scheduler. The Host Controller on the hosts creates and manages a Processing Element Controller (PEC) to execute each PE.

4.5.1 Deploying Operators to specific nodes

There will be circumstances where you want to dedicate Operators to specific nodes in the Instance. For example:

1. You have a Source Operator that is receiving data streamed to a particular IP address.
2. You have a Sink Operator that needs to output data to a particular machine in the Instance for collection by a downstream system.
3. You have processor intensive Operators, the processing of which should only take place on specific hosts.
4. You have UDOP, which invokes external code, which is only possible from specific hosts.

When this is the case, you can define a *nodepools* section in your application, which names the groups of hosts, and then you can specifically bind particular Operators to run on the nodepool hosts. This can be seen in Example 4-12:

Example 4-12 Nodepool and Operator to Host bindings

```
[Application]
Operatorbindingtest

[Nodepools]
# nodepool with named hosts
nodepool SourceHost[] := ( "RBCompute1" )
# nodepool dynamically allocated from known instance hosts
nodepool ComputePool[5] := ()

[Program]
...
# Bound source
stream VideoSource (schemaFor(VideoData))
:= Source()
["sudp://videoUDPServer:2000/"] {}
-> node(SourceHost)

bundle VideoDescriptions := ()
```

```

# Parallel Bound UDOP
for_begin @n 1 to 5
    stream ProcessedVideo@n(SceneDescription : String)
        := UDOP(VideoSubStream@n)
            [ "describeVideoScene" ]
            {}
        -> node(ComputePool,@n)
for_end

# Sink is bound to same host and Source

Null := Sink(VideoDescriptions[:])
    ["file:///video_descriptions.csv", csvformat, nodelays]
    {}
-> nodeFor(VideoSource)

```

Example 4-12 on page 145 shows that:

- ▶ You can define nodepools by naming the specific hosts (nodepool SourceHost[] := ("RBCompute1") in the example).
- ▶ You can define nodepools by specifying the size of the pool required (nodepool ComputePool[5] := () in the example).
- ▶ You can assign Operators to any Host in a nodepool (→ node (SourceHost) in example).
- ▶ You can assign Operators to specific hosts in a nodepool (→ node (ComputePool, @n) in the example).
- ▶ You can colocate Operators with other Operators (→ nodeFor(VideoSource) in the example colocates the Sink with the Operator that produces the VideoSource stream, in this case, this is the Source in the example).

Although not shown in the example, you can also isolate an Operator so that it will not share its Host with other PEs (see the appropriate product documentation for details of this option).

4.6 Performance engineering

Streams performance is a case of supply and demand. Computer hardware supplies compute resources for applications to use, including processors, memory, disk I/O, or network I/O. Applications executing on the hardware demand compute resources, access to the processor for a period of time, use of memory, network I/O, and disk I/O.

Performance problems of course will arise when the demand is greater than the supply. Figure 4-7 gives an illustration of a theoretical performance problem. There are three components shown:

- ▶ The hardware's capacity to supply resources at a particular rate. Here the tpmC number is a server benchmark defined by the Transaction Processing Performance Council (TPC).
- ▶ The resource demand of the Streams Application for performing particular stream processing.
- ▶ The workload, that is, the volume of Tuples that need to be processed in a period of time.

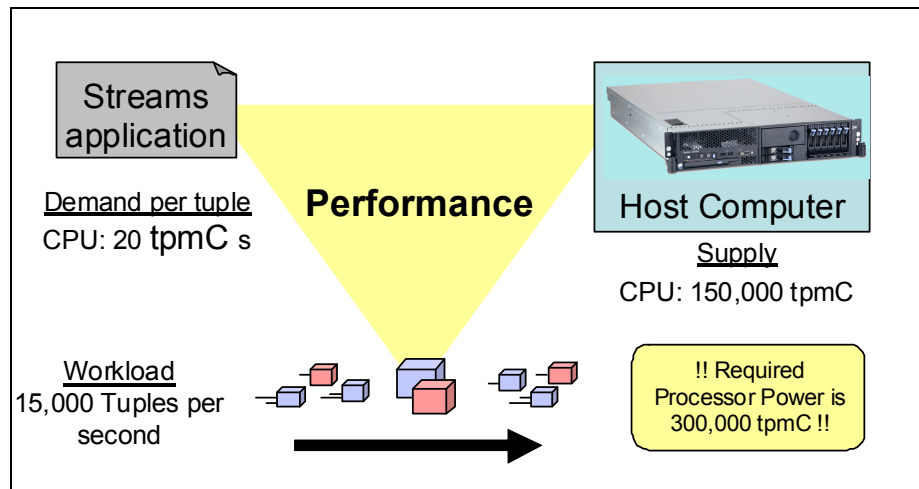


Figure 4-7 Performance is a combination of supply, demand, and workload

In Figure 4-7, we can see that the application shown will require 300,000 tpmC to achieve the specified workload (20 tpmC seconds per Tuple * 15,000 Tuples per second = 300,000 tpmC). So this configuration will result in a performance problem because the system cannot achieve its throughput targets with a single Host computer. There will also be a supply and demand for network bandwidth, which is not shown in this simple illustration.

As Streams is a platform targeted at high volume processing, you need to have a strong focus on defining, testing, and optimizing performance to achieve high performance targets. It is common for newly developed applications to perform sub-optimally on a platform and to require some tuning or refactoring to achieve the performance targets.

The following process is suggested to identify performance problems and improve performance:

1. Define performance targets.
2. Establish access to a suitable environment.
3. Measure performance (throughput and latency) and compare it to the target.

If performance is below the target:

1. Identify the limiting resource, that is, the dominant bottleneck.
2. Identify the source of the bottleneck.
3. Remove the bottleneck.
4. Repeat from step 3 until the performance target is met.

When testing a system for performance, it is important to keep clear targets in mind. It is always possible to spend more effort to optimize an application more and more to make it run faster. However, after you have achieved your performance target, you no longer need to spend this effort. This method adopts a pragmatic approach to achieving performance, which only needs to improve to meet the target workload of the system.

4.6.1 Define performance targets

The driving force between this performance improvement method is to understand the required performance targets. For a Streams system, performance targets can be characterized as follows:

- ▶ Define the expected volumes of inputs to the system (the Source Operators).
- ▶ Define the expected volumes of outputs from the system (the Sink Operators).
- ▶ Define the desired response time requirements between receiving a Tuple (or a set of Tuples) and generating the related output.

These targets have been discussed as part of designing the Streams Application in 3.1.6, “Performance requirements” on page 87 and 4.3, “Sizing the environment” on page 136. In addition to the expected volumes at the current point in time, you should also consider projected growth over the lifetime of the system.

In addition to average rates of throughput, it is important to consider peaks in the workload, including a review of the periodic changes in throughput over time. Volumes may change regularly based on the time of the day, week, month, or year.

4.6.2 Establish access to a suitable environment

When measuring the performance of a Streams system, you need to monitor a running, loaded system and assess the achieved throughput and latency of the application and the resource utilization (CPU, memory, disk I/O, and network I/O) of the computers that comprise the Streams Runtime.

It is possible to establish a performance test environment, separate from the live environment, to measure application performance. It is also possible to measure performance of live environments to confirm that performance targets are being achieved and to allow you to accurately plan for system expansion as the system volumes grow.

There are advantages and disadvantages to testing in a performance test or live environment. Consider the following items:

- ▶ Hardware cost

To be truly effective, a performance test system needs to be a replica of the hardware of the live environment. In cases of large hardware clusters, the cost of a replica system would be significant.

As a compromise, it could be possible to share some of this test hardware with the live environment so that it is used for testing when the live environment is under a smaller load. You could have a test environment that is a smaller subset of the live environment. Caution is needed, though; a smaller test environment may be unable to reproduce the same performance characteristics that will be seen in a live environment, so the objectives of performance testing will be compromised.

- ▶ Setup cost

- In a performance test environment, you need to set up artificial load injectors to simulate the inputs to Streams and to generate test data to inject. You may also need to simulate the processing of external systems, particularly downstream systems that consume the system output.
- In a live environment, the real sources are providing input, you interface with live external systems, and the real systems are consuming the output streams, so there are minimal setup costs.

- ▶ Freedom to change

- In a test environment, you have freedom to change the application, you can start and stop applications and PEs, and you can fuse Operators in different combinations to assess the optimal performance configuration.
- In a live environment, the system is actively processing data and generating results, so you will have a limited ability to start, stop, and modify the running applications.

- ▶ Freedom to inject load
 - In a test environment, you have the freedom to inject load in whatever profiles and volumes you like, and you can overload the system to see what happens and inject large quantities of unusual events to see if the processing is impacted.
 - In a production environment, you are limited to observing the actual load traversing through the system, and you have limited or no freedom to inject more or less load to stress the system.
- ▶ Monitoring tools
 - In a test environment, you can use more invasive monitoring tools, such as profilers and debuggers, which might impact the system and slow down performance or impact the processing results. These more invasive tools may be invaluable in diagnosing performance problems.
 - Monitoring tools must not interfere with operation in a live environment. This is likely to rule out the use of debuggers and profilers, but should allow the general use of operating system tools to monitor system resources, such as processor and network activity.
- ▶ Proactive optimization
 - In a test environment, you can be more proactive in your approach to testing performance. You can try out different performance scenarios to measure the throughput and you can try out alternative configurations and combinations of input to diagnose and fix performance problems.
 - In a live environment, you are limited to observing the actual performance of the live system. You can diagnose performance problems in a live environment, but this is a more reactive process.

In summary, the trade-off is between cost and effectiveness of testing. Performing performance testing in a live environment is cheaper, but testing in a dedicated performance test environment is more effective in optimizing applications and achieving performance targets.

4.6.3 Measure the performance and compare it to the target

The aim of this step is to measure the throughput and response time from a Streams Application when the system is under a full load, and compare the performance to the target.

In a test environment, you will need load injection capability and a test harness that will generate streams of test data for consumption by the Stream Application under test. You need to make multiple measurements to make sure that you have a reasonable average.

You need to measure the following items:

- ▶ Throughput: Streams performance counters can be used to measure the number and size of Tuples for all Streams Processing Elements.
- ▶ Latency: If this is a requirement, then you should time stamp Tuples when they are ingested into Streams and compare the times when their processing is complete.
- ▶ Resource utilization: Use operating system resource management tools to measure the CPU load, memory utilization, disk I/O, and network I/O of each Host of the Streams runtime cluster. A tool such as Ganglia (<http://ganglia.sourceforge.net/>) could help enable a distributed, resource monitoring capability for clusters of hosts that is compatible with Linux.

Figure 4-8 shows a Streams Application under evaluation in a performance test environment. Here we are artificially generating and injecting load, and simulating systems to consume the output streams from the application. We are also monitoring the throughput using the management Host's Performance Counters and the resource usage of each Host.

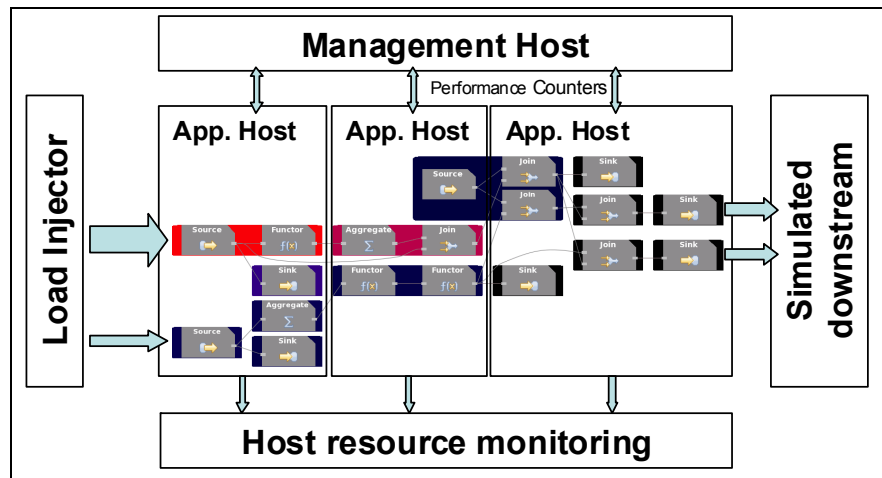


Figure 4-8 Measuring performance of a Streams Application

If the performance is consistently better than the target, then no optimization is required. It is important to take a number of measurements, as all performance measurements will have some variation. The measurements will have less variability in a dedicated environment with a large, steady load as opposed to a shared environment with a fluctuating load.

Measuring throughput

Streams Applications automatically maintain metrics about the throughput and CPU usage of the processing entities. You can use a number of tools to display these metrics.

You can use the **streamtool runcmd perf_client** command to show performance information for a running Streams Application, as shown in Example 4-13.

Example 4-13 Using streamtool runcmd perf_client to show performance information

```
$ streamtool runcmd -i inst perf_client --pe=3 -L --count=3 --loop=3  
txtuples: 27714, rxtuples: 28256, txbytes: 770590, rxbytes: 2489470  
txtuples: 32624, rxtuples: 33165, txbytes: 907139, rxbytes: 2921892  
txtuples: 37587, rxtuples: 38128, txbytes: 1045215, rxbytes: 3359221
```

The IBM InfoSphere Streams Studio provides a number of views to display performance metrics. Figure 4-9 shows the Streams Live Graph monitoring a Streams Application. You can see that Operators are grouped by processing entity and the processing entities are colored to show the level of throughput for each PE. The red colored PEs have a high load and the blue and black PEs have a lower throughput at this point in time.

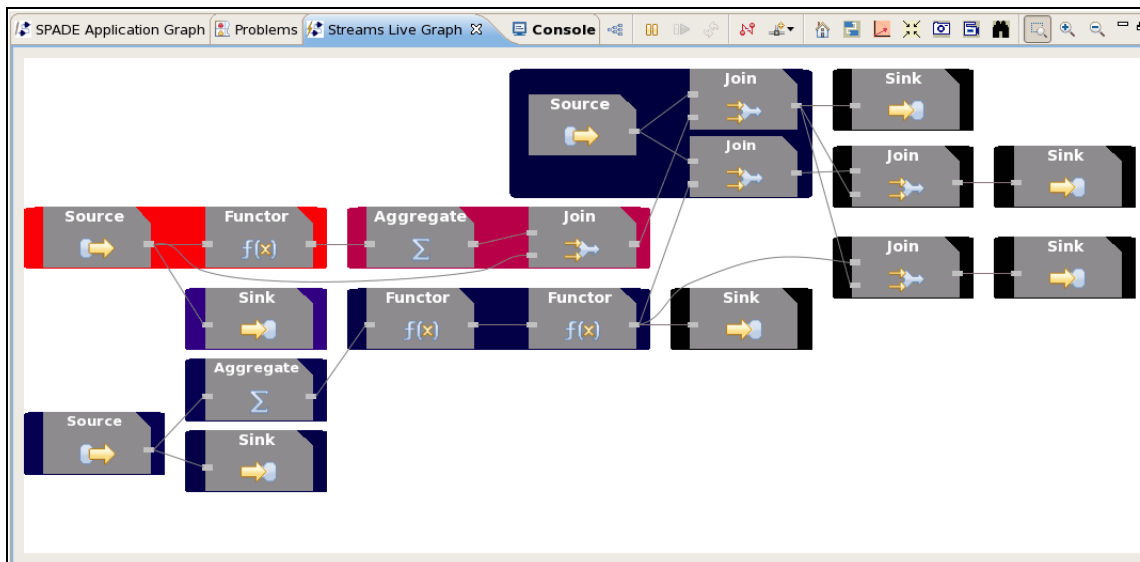
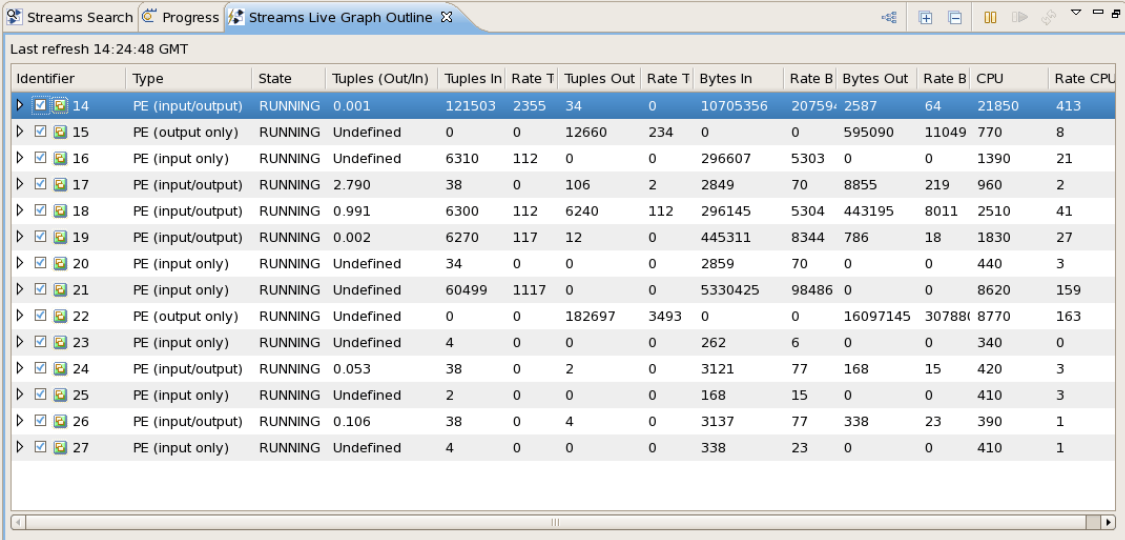


Figure 4-9 Streams Live Graph monitoring an application

Figure 4-10 shows the Streams Live Graph Outline View for an application. The Streams Live Graph Outline View shows the performance statistics for the individual PEs in the application; this includes volumes in and out of each PE (count of Tuples and size in bytes) and a level of CPU consumption for each PE.



The screenshot shows a software window titled 'Streams Live Graph Outline'. At the top, there are tabs for 'Streams Search', 'Progress', and 'Streams Live Graph Outline'. Below the tabs, it says 'Last refresh 14:24:48 GMT'. The main area contains a table with 14 rows, each representing a PE (Processing Element). The table has columns for Identifier, Type, State, Tuples (Out/In), Tuples In, Rate T, Tuples Out, Rate T, Bytes In, Rate B, Bytes Out, Rate B, CPU, and Rate CPU. The first row (PE 14) is highlighted in blue. The other rows are in white with alternating shades of gray for the data columns.

Identifier	Type	State	Tuples (Out/In)	Tuples In	Rate T	Tuples Out	Rate T	Bytes In	Rate B	Bytes Out	Rate B	CPU	Rate CPU
14	PE (input/output)	RUNNING	0.001	121503	2355	34	0	10705356	20759	2587	64	21850	413
15	PE (output only)	RUNNING	Undefined	0	0	12660	234	0	0	595090	11049	770	8
16	PE (input only)	RUNNING	Undefined	6310	112	0	0	296607	5303	0	0	1390	21
17	PE (input/output)	RUNNING	2.790	38	0	106	2	2849	70	8855	219	960	2
18	PE (input/output)	RUNNING	0.991	6300	112	6240	112	296145	5304	443195	8011	2510	41
19	PE (input/output)	RUNNING	0.002	6270	117	12	0	445311	8344	786	18	1830	27
20	PE (input only)	RUNNING	Undefined	34	0	0	0	2859	70	0	0	440	3
21	PE (input only)	RUNNING	Undefined	60499	1117	0	0	5330425	98486	0	0	8620	159
22	PE (output only)	RUNNING	Undefined	0	0	182697	3493	0	0	16097145	30788	8770	163
23	PE (input only)	RUNNING	Undefined	4	0	0	0	262	6	0	0	340	0
24	PE (input/output)	RUNNING	0.053	38	0	2	0	3121	77	168	15	420	3
25	PE (input only)	RUNNING	Undefined	2	0	0	0	168	15	0	0	410	3
26	PE (input/output)	RUNNING	0.106	38	0	4	0	3137	77	338	23	390	1
27	PE (input only)	RUNNING	Undefined	4	0	0	0	338	23	0	0	410	1

Figure 4-10 Streams Live Graph Outline monitoring an application

Gradually increase the load

You should start off testing by injecting a low rate of incoming Tuples and increase this load (hopefully up to the target volumes). Measure the achieved throughput and latency of the system as the load increases, and plot the injected load against the achieved throughput and latency for analysis.

In Figure 4-11, we show a typical graph of throughput and latency as we increase the rate at which streams Tuples are injected into the system:

- ▶ Initially, the increasing rate of incoming Streams volumes produces a proportional increase in throughput and the latency of each Tuple is relatively constant (and low).
- ▶ At a particular rate of incoming Tuples, the application reaches a saturation point due to a bottleneck in the system. Increasing the rate of Tuples does not increase the throughput, but instead the latency increases; each Tuple queues up in the Streams run time and takes longer to be processed.
- ▶ Increasing the input rate still further may result in reaching a buckle point, at which point the application may fail or start to lose data.

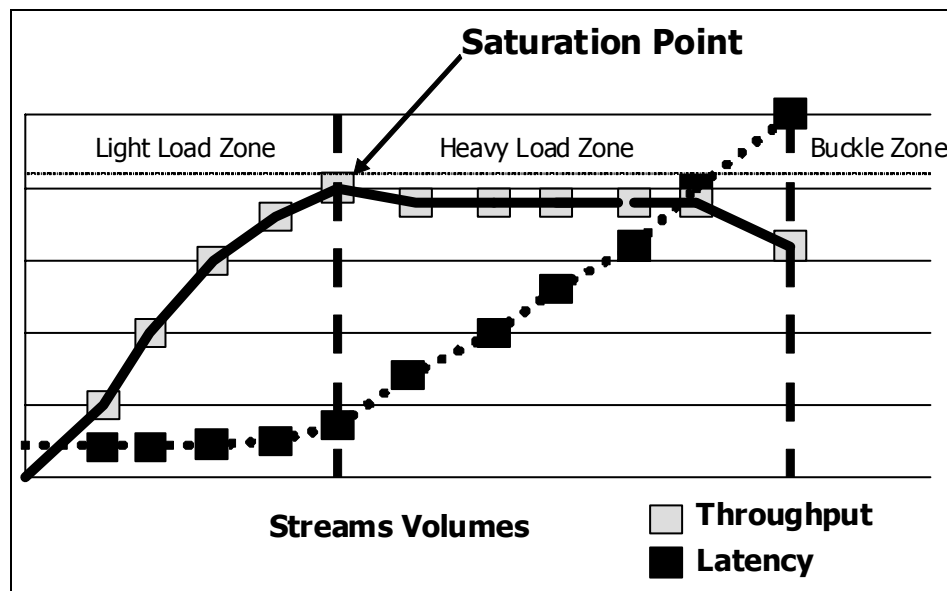


Figure 4-11 Measuring throughput and latency while increasing Streams Volumes

If the system achieves its throughput targets, then you will be able to reach the target load injection without reaching the saturation point.

If the system does not achieve its throughput targets, then you should run the application with profiling enabled (as described in 4.9, “Environment profiling” on page 180). The profiling data should be used by the compiler to intelligently fuse the Operators into PEs, and you should retest your application with this new set of generated PEs.

If the system does not achieve its throughput targets after profiling and regenerating PEs, then you will need to locate the dominant bottleneck and determine which Processing Elements are limiting the throughput of the application.

4.6.4 Identify the dominant bottleneck

If performance is below the target, then you need to discover what is limiting the performance of the application. Typically, in every application at any point in time, there will be a particular processing entity that is limiting throughput; we call this processing entity the *dominant bottleneck*.

If you are running with intelligent PE fusing (through the compiler) and you still have a performance problem, then you should look at the resource utilization of the hosts in the Streams run time:

- ▶ Are any of the hosts running at over 90% CPU utilization? If so, then this is likely to be the host with the dominant bottleneck.
- ▶ If no hosts are running at 100% CPU utilization, are there any hosts with significant amounts of wait time in their CPU? This is an indication that the PEs on this Host are throttled by writing to disk or to network. Either of these situations could result in a bottleneck and should be further investigated.
- ▶ You will be injecting multiple streams of data into a Streams Application; only some of these streams will be saturated by the dominant bottleneck. Plotting the throughput graphs will aid you in identifying which Source streams are saturated and which are not; this will help you identify the dominant bottleneck.

4.6.5 Remove the bottleneck

After you have identified the Host or the Processing Elements responsible for the dominant bottleneck, then you can increase the throughput of the limiting components.

Remember that performance is an issue of supply and demand; to increase the throughput of the system, you can increase the supply of the resource by providing more or faster computers, or you can reduce the demand of resource by decreasing or redistributing the processing performed by the application:

- ▶ If there are multiple PEs running on the bottleneck Host, you can identify the PE with the highest load and use the nodepool configuration (4.5.1, “Deploying Operators to specific nodes” on page 145) to run this PE on its own Host.

- ▶ You may supply more resources for the resource intensive PEs, that is, provide a machine with faster or more processors, faster disks, or a faster network connection.
- ▶ You can split the processing of the resource intensive PE, that is, break it down into separate Operators running in parallel or pipelined mode.

After you have taken steps to remove the bottleneck, you should rerun your performance tests and check that your changes have made matters better (revert the changes if they make things worse) and repeat the process of testing and optimization until the system achieves its performance targets.

4.7 Security

In this section, we present an overview of the security features of Streams. The security levels supported in Streams are as described in the following sections, and depicted in Figure 4-12 on page 157.

Secure access

Installation and normal operation of Streams establishes a number of security aspects to allow hosts to securely access each other:

- ▶ SSH communication between Streams Hosts allows remote execution of administration scripts.
- ▶ PAM or LDAP authentication allows user authentication between runtime services.

Instance and job control

It is possible to configure a security policy for a Streams Instance that defines the authentication, authorization, and auditing for users of the Instance. This controls which users can start and stop Instances, submit applications, and so on.

A Streams Application is submitted to run as a Job on the distributed hosts of the Streams Instance. This causes the execution of Processing Elements, which, as modules of compiled Streams code, can perform a wide range of actions.

Processing Element containment

If you want to constrain what actions the Processing Elements can perform to interact with the outside world, such as file system and network access, then you should use aspects of the secure operating system SELinux. You can define a number of security domains within which the PEs execute, and restrict the access of PEs to system resources, thus reducing the chance of any malicious or accidental impacts. This process is called Processing Element containment.

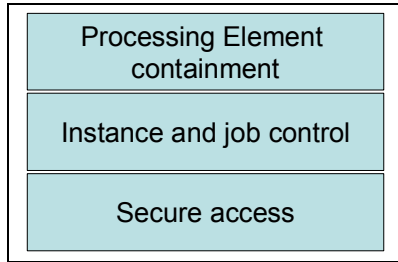


Figure 4-12 Streams security levels

Security walkthrough

Figure 4-13 on page 158 (and the remainder of this section) gives an overview of the main security aspects of the Streams run time through the process of creating an Instance and running Streams Applications.

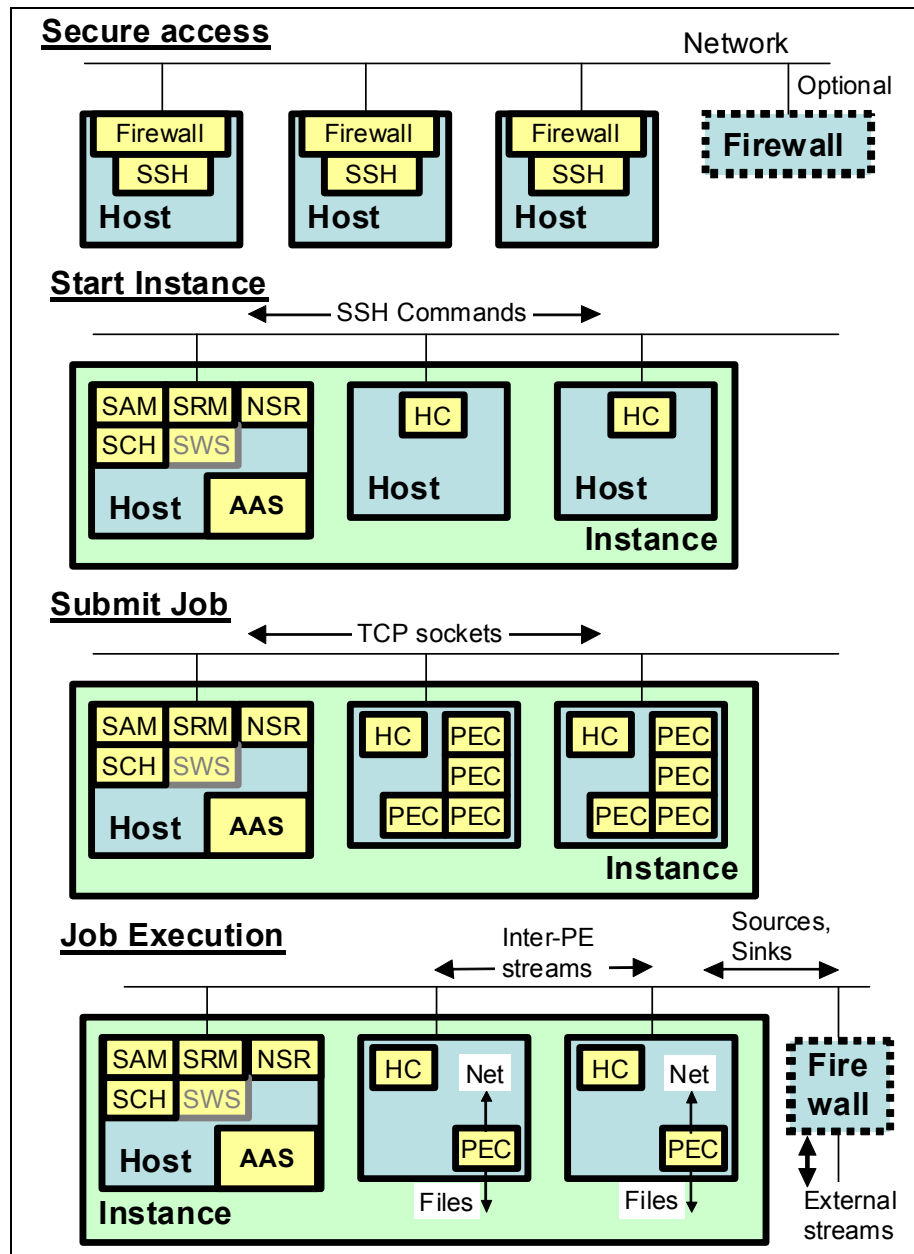


Figure 4-13 Security walkthrough

Where:

- **Secure Access:** After installation, you have established and exchanged SSH key certificates between hosts in the Streams cluster. This allow secure, remote execution of commands by the Streams owner across the distributed hosts.

You may also enable the operating system firewall and a network firewall to restrict network traffic to the known Streams communication mechanisms between hosts and to external networks (See 4.7.1, “Secure access” on page 160 for more details).

- **Start Instance:** The Streams administration tools allow you to configure and start an Instance. You need to specify a security policy to be enforced by the AAS service for authentication and authorization of the Instance operations. All the Streams runtime services refer to AAS for authorization to perform their operations.

The SSH protocol is used to start the Instance services on the distributed hosts. SSH and firewall security will apply for this and the following activities, but these are not shown in the illustration for reasons of clarity.

Note: In the illustration, we established a management Host with all the management services and two application hosts that run the Host Controller (and subsequently Processing Element containers). This separation of concerns is recommended for security purposes, to avoid the risk of rogue PEs impacting management services.

- **Submit Job:** You invoke the Streams Application Manager (SAM) service to run a Streams Application Job in the Streams Instance. The SAM will authorize this job through the AAS service to ensure that the user submitting the job has the appropriate privileges, and if so, will send the job to the Host controllers (HC) on each application Host to run Processing Elements for the application job. Each Processing Element will run in its own Processing Element container (PEC).
- **Job Execution:** Each PEC executing as part of the application job can interact with the file system of the Host computer and can send and receive network traffic either with other PEs or with external sources of Streams, sources, and Sinks from outside this network. The PEs are compiled code, and so can perform a wide range of activities in the file system and networks. You can restrict the potential activities of the PEs by applying operating system (SELinux) policies and types.

4.7.1 Secure access

There are four aspects of secure access that we cover in this section:

- ▶ Enabling SELinux before installation
- ▶ Configuring SSH access after installation
- ▶ Generating Rivest, Shamir, and Adleman (RSA) keys after installation
- ▶ Configuring firewall access between hosts

SELinux

Note: For any production server, we recommend that Streams be installed with SELinux enabled. SELinux is required for some of the advanced security aspects recommended for Streams production servers, and it is preferable to enable SELinux before the installation rather than afterwards.

Before installation, check the SELinux status using the **sestatus** command (see Example 4-14). Ensure that:

- ▶ The SELinux status is set to enabled.
- ▶ The current mode is set to enforcing or permissive.
- ▶ The policy is set to targeted.

Example 4-14 Checking SELinux status

```
(Run as root)
# sestatus
SELinux status:                enabled
SELinuxfs mount:              /selinux
Current mode:                  enforcing
Mode from config file:         enforcing
Policy version:                21
Policy from config file:       targeted
```

If SELinux is disabled, or you want to change the SELinux policy, then use the Linux security GUI to change the SELinux mode. Perform the following steps:

1. From the System menu of the Red Hat Linux desktop, select **Administration** and click **Security Level and Firewall** to display the Security Level Configuration window (you will be prompted to enter the root password).
2. Click the **SELinux** tab.
3. In the SELinux Setting window, select **Enforcing** (see Figure 4-14 on page 161).

After enabling SELinux, you will need to restart the system. After the reboot, SELinux will apply file system labels to all files, which may take a significant amount of time.

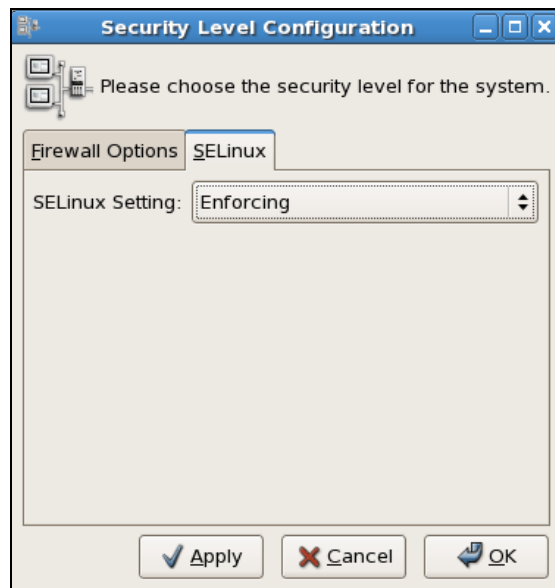


Figure 4-14 Enabling SELinux using the security GUI

SSH configuration

After the Streams installation is complete, you need to establish secure, password-less SSH access between the hosts for the installation user and for users that you will be using to run Streams administration commands.

Example 4-15 shows how to generate SSH keys for Streams users. The **ssh-keygen -t dsa** command is used to generate keys, and the generated keys are then appended to the `~/.ssh/authorized_keys` file to allow password-less SSH logon. The logon is tested by the **ssh localhost** command, but you should not be prompted for a password as part of this command.

Example 4-15 Generating SSH keys for the execution of Streams commands and scripts

Generate SSH Keys:

```
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key (/home/streamsadmin/.ssh/id_dsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/streamsadmin/.ssh/id_dsa.
```

Your public key has been saved in /home/streamsadmin/.ssh/id_dsa.pub.
The key fingerprint is:
90:3a:ba:cd:3f:dc:e3:8b:70:67:99:91:80:85:42:f8
streamsadmin@redbookstreams

Now use the keys to authorized secure logon:

```
$ cd ~/.ssh  
$ cat id_dsa.pub >> authorized_keys  
$ chmod 600 *
```

Test the ssh:

```
$ ssh localhost  
Last login: Wed Mar 24 14:24:51 2010  
InfoSphere Streams environment variables have been set.
```

RSA configuration

As part of the installation, Streams will establish RSA authentication keys for the installation user. These keys allow Streams to authenticate you as a Streams Instance user without needing to enter a password.

If you need to regenerate keys, or establish them for a different user, you can use the **streamtool genkey** command to generate a public and private key. This places the keys into the ~<user>/.streams/key/ directory, as shown in Example 4-16.

Example 4-16 Generating RSA keys for authentication between runtime services

```
$ streamtool genkey  
CDISC0013I The private key file  
'/home/streamsadmin/.streams/key/streamsadmin_priv.pem' was created.  
CDISC0014I The public key file  
'/home/streamsadmin/.streams/key/streamsadmin.pem' was created.
```

Firewall configuration

If the security requirements call for the firewalls to be configured on the host operating systems or between hosts, then the following communications should be enabled between hosts, and blocked from unauthorized external access:

- ▶ SSH communication between runtime hosts.
- ▶ Communication between management services using ports on the local port range (TCP/IP port numbers automatically assigned by the host machine).

- ▶ TCP communication between Processing Elements.
- ▶ HTTP/HTTPS connection from an administration browser to the SWS service running on a management Host.
- ▶ All authorized communication protocols between the application and its Source streams.
- ▶ All authorized communication protocols between the application and its downstream systems.
- ▶ All authorized communication protocols between the application and any external systems, such as databases or external analytics services.

4.7.2 Security policies: Authentication and authorization

Each Streams Instance has an associated security policy defining which users can view and alter objects in the Streams Instance. The security policy is enforced by the Authentication and Authorization Service (AAS).

The security policy defines three distinct aspects of security:

- ▶ Authentication module: Establishing the credentials of a user, confirming that they should be treated as an authentic user of the Streams Instance.
- ▶ Authorization policy: Given a user's established credentials, determine which operations they can perform on which objects of the Streams Instance.
- ▶ Audit policy: Defines whether you keep an audit log of the security events for the Streams Instance.

When you create an Instance, you can define the security policy by specifying the `--sectemplate` option, as shown in Example 4-17.

Example 4-17 Defining the security policy when creating an Instance

```
$ streamtool mkinstance -i inst --sectemplate private --numhosts 1
CDISC0040I Creating Streams instance 'inst@streamsadmin'...
CDISC0001I The Streams instance 'inst@streamsadmin' was created.
```

The following security policy templates are predefined with Streams:

- ▶ Private: Provides authentication based on operating system login and RSA keys. There is no audit file created and authorization only allows the Instance creator to use the Instance.
- ▶ Private-audit: Provides authentication based on operating system login and RSA keys. An audit file is created and maintained. Authorization only allows the Instance creator to use the Instance.

- ▶ **Shared:** Provides authentication based on operating system login and RSA keys. There is no audit file created and authorization allows all users in the admin group and the users group to use the Instance.
- ▶ **Shared-audit:** Provides authentication based on operating system login and RSA keys. An audit file is created and maintained. Authorization allows all users in the admin group and the users group to use the Instance.

As can be seen, the private templates are intended for use for Instances where the Instance creator is the sole user. The shared templates are intended for Instances with multiple users.

Defining the security policy

The valid security policies are defined in the following security configuration file:

```
<Streams Install Dir>/etc/cfg/config-templates.xml
```

This configuration file can be modified to change the behavior of the predefined security policies, or to add your own policy templates. An abridged version of the policy template configuration file is shown in Example 4-18.

Example 4-18 Security policy template configuration file (abridged)

```
<config-templates xmlns:cfg="http://www.ibm.com/config-templates.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ibm.com/config-templates.xsd
config-templates.xsd">
  <plugins>
    <authentication>
      <plugin name="ldap">
        ....
      </plugin>
      ....
    </authentication>
    <auditlog>
      <plugin name="fileAudit">
        </plugin>
      </auditlog>
    </plugins>
    <policies>
      <authorization>
        <policy name="private" auto-public-keys="true">
          .....
        </policy>
        ....
      </authorization>
```

```

</policies>
<templates>
    ....
    <template name="example-template">
        <authentication ref="ldap" />
        <auditlog ref="fileAudit" />
        <authorization ref="private" />
    </template>
</templates>
</config-templates>

```

Where:

- ▶ The highlighted section starting `<template name="example-template">` defines the security policy template.
- ▶ The security policy template references three other policy modules: authentication (“ldap” in this example), auditlog (“fileAudit” in this case), and authorization (“private” in this case).
- ▶ The details of these policy modules are also defined in the config file; the details are not shown in this example.

Authentication

The process of authentication allows you to confirm that a user of the system is a recognized, approved user of the Streams Instance and you are able to establish credentials for the user.

The Streams run time can use two different authentication methods: Pluggable Authentication Module (PAM) or Lightweight Directory Access Protocol (LDAP). The choice of authentication method is made while creating the Streams Instance, as part of the security policy template.

The security policy template references a named authentication plug-in configuration, which is defined in the security configuration file (`<Streams Install Dir>/etc/cfg/config-templates.xml`).

Example 4-19 shows the section of the security configuration file that defines the authentication method details. This file should be modified or expanded to define specific authentication mode parameters and user specific security templates.

Example 4-19 Authentication plug-in configuration

```

....
<plugins>
    <authentication>
        <plugin name="pam">

```

```

    <fragment>
      <![CDATA[
        <pam>
          <service>%service%</service>
          <enableKey>%enableKey%</enableKey>
        </pam>]]>
    </fragment>
    <parameters>
      <parameter name="pamService" substituteFor="%service%">
        <default>login</default>
      </parameter>
      <parameter name="enableKey" substituteFor="%enableKey%">
        <default>true</default>
      </parameter>
    </parameters>
  </plugin>
<plugin name="ldap">
  <file
name="%STREAMS_INSTALL_OWNER_CONFIG_DIRECTORY%/ldap-config.xml"></file>
</plugin>
</authentication>

```

You can see two authentication configurations in the example file:

- ▶ pam: This is configured to enable the PAM authentication mode and has two parameters:
 - Service (in this case, we are using the operating system login credentials for authentication)
 - enableKey (in this case, we are allowing authentication using RSA generated keys).
- ▶ ldap: This is configured to use the LDAP authentication mode and uses the environment variable STREAMS_INSTALL_OWNER_CONFIG_DIRECTORY to get parameters from the ldap-config.xml file in the local config directory of the Instance owner. In practice, this file needs to be created in <install-owner>/.streams/config/.

The following parameters need to be defined for the LDAP protocol:

- serverUrl: The host name and port number of the LDAP server
- userDnPattern: The pattern for referencing a user in the LDAP database
- groupObjectClass: The class holding the LDAP group entries
- groupSearchBaseDn: The root search location for finding group names

- `groupAttributeWithUserNames`: The Attribute in a group that contains associated users
- `userAttributeStoredInGroupAttribute`: The Attribute in a user record that is stored in a group record

For both authentication methods, you can define the session timeout for a user session using the `SecuritySessionTimeout` property. This means that a user session will expire after the specified period of inactivity. The property can be defined when creating an Instance or can be set after the Instance is created. Example 4-20 shows how to change the property for an existing Instance.

Example 4-20 streamtool setproperty command to set the security session timeout

```
$ streamtool setproperty -i inst SecuritySessionTimeout=3600
CDISC0008I The 'SecuritySessionTimeout' property has been set to '3600'
(previous value was '<undef>').
```

Note: A change to the session timeout parameter will only take effect after you restart the Instance.

Use RSA keys

The Streams security service, AAS, requires that users authenticate when using specific streamtool commands (`submitjob`, `canceljob`, `cancelpe`, `restartpe`, `setacl`, `lsjobs`, and `lspes`). The `enableKey` option in the authentication security policy simplifies the user authentication by allowing RSA keys to be generated and used to authenticate the user. See 4.7.1, “Secure access” on page 160 for details about how to generate these keys.

Authorization

An authorization policy consists of a set of Access Control Lists (ACLs). These ACLs define the operations that each authenticated user can perform on Streams objects.

There are two predefined authorization policies:

- ▶ Private: Intended for use for Instances with a single user (the Instance owner)
- ▶ Shared: Intended for Instances with multiple users

Streams objects and operations

The following Streams objects are controlled by the authorization policy:

- ▶ Instance: The overall administrative scope
- ▶ Config: The Instance configuration
- ▶ Hosts: The set of hosts for the Instance, and services on those hosts

- ▶ jobs: The list of jobs submitted for the Instance
- ▶ job_<x>: A specific job and its Processing Elements (PEs).
- ▶ jobs-override: Controls the ability to override Host load protection when submitting jobs.

The following operations on the objects are controlled:

- ▶ Instance: Log in and change
- ▶ Config: View and change
- ▶ Hosts: View, access, add, remove, add services, and remove services
- ▶ jobs: List and submit jobs
- ▶ job_<x>: Get data, send data, list information, restart job, restart PE, stop job, and stop PE
- ▶ jobs-override: Override the Host load protection feature.

Access Control List permissions

The ACLs allocate permissions to Streams objects for specific users, restricting which operations can be performed by that user. The following permissions may be allocated:

- ▶ Read (r)
- ▶ Write (w)
- ▶ Search (s)
- ▶ Add (a)
- ▶ Delete (d)
- ▶ Own (o)

Table 4-1 shows which ACL permissions apply to which Streams objects, and which operations on the objects. For example:

- ▶ You can allocate two permissions for an Instance object: search and own. The search permission enables a user to log in to the Instance and the own permission enables a user to modify ACL permissions for the Instance.
- ▶ You can allocate three permissions for config object: read, write, and own. The read permission enables a user to view the config object, the write permission enables a user to change the config object, and the own permission enables a user to modify ACL permissions for the Instance.

Table 4-1 Access Control Lists permissions for Stream object operations

	Read	Write	Search	Add	Delete	Own
Instance	-	-	Login	-	-	Perm

	Read	Write	Search	Add	Delete	Own
Config	View	Change	-	-	-	Perm
Hosts	View	-	Access	Add Host and add service	Remove Host and remove service	Perm
Jobs	-	-	List	Submit	-	Perm
Job_<x>	Get data	Send data	List data	Restart PE	Stop job and stop PE	Perm
Jobs-override	-	-	-	-	-	Perm

The ACL permissions are initially set at the time of Instance creation, according to the authorization policy defined. You can use the **streamtool** command to display and modify these permissions:

Example 4-21 shows how to display the permissions for a Streams Instance. In the example, we use the **streamtool getacl** command to display the permissions for the Instance and config objects in the inst Instance:

- ▶ Instance: The streamsadmin user has search and own permissions (user:streamsadmin:--s--o), so it is the only user that can log in to the Instance and change its permissions.
- ▶ Config: The streamsadmin user has read, write, and own permissions (user:streamsadmin:rw---o), so it is the only user that can view and change the Instance configuration and change its permissions.

Example 4-21 Displaying Access Control List permissions

```
$ streamtool getacl -i inst instance config
# object: instance
# parent:
# owner: nobody
# persistent: yes
user:streamsadmin:--s--o

# object: config
# parent: instance
# owner: nobody
# persistent: yes
user:streamsadmin:rw---o
```

Example 4-22 shows how to modify the permissions. In the example, we use **streamtool setacl** to add read permissions (+r) for the streamsuser user ID (u:streamsuser) to the Instance configuration (config). After this command completes, streamsuser will be permitted to read the configuration, but will not be able to change it or modify its permissions. In the example, we confirm the ACL by running the **streamtool getacl** command.

Example 4-22 Modifying Access Control List permissions

```
$ streamtool setacl -i inst u:streamsuser+rw config
CDISC0019I The access control list for Stream instance
'inst@streamsadmin' was updated.
```

```
$ streamtool getacl -i inst config
# object: config
# parent: instance
# owner: nobody
# persistent: yes
user:streamsadmin:rw---o
user:streamsuser:r-----
```

Note: The Streams Instance needs to be started to display and modify Access Control Lists permissions. This is because the AAS service manages ACLs; AAS is started with the Instance.

Audit

The security policy defines whether the AAS service will keep an audit log of security events for the streams Instance.

Example 4-23 shows the default audit file configuration (this is part of the security configuration file <Streams Install Dir>/etc/cfg/config-templates.xml). When you create an Instance with the security policy “private-audit” or “shared-audit”, then this configuration will be used, and security events will be logged to the %STREAMS_INSTANCE_DIRECTORY%/audit.log directory; in the case of the streamsadmin user and the ins” Instance, this is /home/streamsadmin/.streams/instances/inst@streamsadmin/config/audit.log.

Example 4-23 Default audit file configuration

```
<auditlog>
  <plugin name="fileAudit">
    <fragment> <![CDATA[
      <file>
        <filename>%auditLogFile%/</filename>
```

```

        </file>]]>
    </fragment>
    <parameters>
        <parameter name="auditLogFile" substituteFor="%auditLogFile%">
            <default>%STREAMS_INSTANCE_DIRECTORY%/audit.log</default>
        </parameter>
    </parameters>
</plugin>
</auditlog>

```

4.7.3 Processing Element containment

The previous sections show how you can configure the Streams run time to allow, but restrict, access between Streams hosts using SSH and firewalls (4.7.1, “Secure access” on page 160) and to allow restricted access to Streams administration objects and their operations using the AAS service for the Streams Instance (4.7.2, “Security policies: Authentication and authorization” on page 163). This includes the ability to submit jobs to run Streams Applications.

Each Processing Element (PE) executing as part of the application job can interact with the file system of the Host computer and can send and receive network traffic either with other PEs or with external sources of Streams, such as sources and Sinks from outside this network.

The PEs are compiled code, and so can perform a wide range of activities with the file system and networks. You can restrict the potential activities of the PEs by applying operating system (SELinux) policies and types in an approach we call *Processing Element containment*.

There are three levels of PE containment that can be applied using SELinux:

- ▶ Unconfined PEs: All Processing Elements run in an unconfined security domain and are allowed to read and write files and send and receive data over the network.
- ▶ Confined PEs: All Processing Elements run in a single confined security domain. The standard Streams policy module only allows limited rights to the PEs and additional privileges need to be enabled by providing your own SELinux policy modules.
- ▶ Confined and Vetted PEs: Processing Elements are run in the confined security domain. Trusted PEs can be marked as “Vetted” and will run in a different, vetted security domain. You need to provide your own SELinux policy modules to enable PEs in either domain to access external data.

SELinux basics

SELinux is a security architecture built into Red Hat Enterprise Linux 5 to provide Mandatory Access Control (MAC) security. MAC allows a security administrator to constrain the access of processes and threads running in Linux to the files and network communication ports.

The mandatory aspect to MAC refers to the fact that the security policy is controlled by a security administrator and normal Streams users are not able to override the policy. This contrasts with the normal Linux access control mechanisms where users may own their files and can define the security Attributes of these files to allow access to other users.

SELinux defines the access privileges of the users, processes, and files on the system. Access between users and processes to files is managed by the SELinux Security Server, which determines whether permission for the access should be granted.

The strictness with which SELinux enforces the defined security policies can be controlled by two main options (These are specified in the configuration file `/etc/sysconfig/selinux`):

- ▶ **SELINUX:** The main state of SELinux, this can be:
 - Disabled: SELinux is fully disabled.
 - Permissive: SELinux is enabled but does not enforce the defined policies.
 - Enforcing: SELinux is enabled and enforces the defined policies. This is recommended for Streams production servers.
- ▶ **SELINUXTYPE:** Defines which security policy is enforced. This policy can be:
 - Targeted: The SELinux security is only enforced for targeted processes. Enforcement for processes can be enabled or disabled by setting Boolean values. For example, setting the `streams_confined` Boolean to true enforces security restriction on Streams Processing Entities. This is the recommended level for Streams production servers
 - Strict: The SELinux policy is enforced against all processes and every action is considered by the policy enforcement server.

An SELinux policy allocates *types* to file objects (including network sockets). Files with similar security requirements are allocated the same type. The policy also allocates *domains* to processes and to users. The policy then defines the actions that a process in a named domain can perform on a file of a named type. If no rule is defined to permit an action, then the action is denied by default.

A security policy is constructed from separate *policy modules*. A policy module defines a set of allowed actions from named domains to named types. You can define and install your own additional modules to allow specific actions.

Processes and users that are in the *unconfined_t* domain have no restrictions and use the standard Linux model of user permissions.

Unconfined PEs

When running in SELinux Enforcing and Targeted mode, the `streams_confined` Boolean determines in which domain the Processing Entities will run. If `streams_confined` is set to false, Processing Entities run in the `streams_unconfined_t` domain.

When running in this unconfined domain, PEs can read and write files from the file system and can send and receive data over the web. However, there are some restrictions, for example, they are not permitted to change Streams installation files to avoid accidental or malicious harm to these files.

You can check the status of the `streams_confined` Boolean using the **getsebool** command, as shown in Example 4-24. In this example, `streams_confined` is false, so all PEs will run in the unconfined domain.

Example 4-24 Checking the streams_confined Boolean in SELinux

```
# /usr/sbin/getsebool streams_confined
streams_confined --> off
```

Confined PEs

If a greater level of control over the actions of PEs is required, you can set the SELinux Boolean `streams_confined` to true and then Processing Entities will run in the `streams_confined_t` domain.

When running in this domain, PEs are subject to the Streams policy module, which prevents them from reading and writing files from the file system and sending and receiving data over the network. You can define your own SELinux policy modules to allow PEs necessary access to specific, appropriate external resources.

Note: The writing of SELinux policy modules is a complex activity which is beyond the scope of this book. For more instructions, see Chapter 4 of the *Streams Installation and Administration Guide* and the *Red Hat Enterprise Linux Deployment Guide*.

You can check the status of the `streams_confined` Boolean using the **getsebool** command. You can change the `streams_confined` Boolean using the **setsebool** command and setting the value to true. These commands are shown in Example 4-25. In this example, `streams_confined` is false; we set it to true to run PEs in confined mode.

Example 4-25 Setting the `streams_confined` boolean in SELinux

```
# /usr/sbin/getsebool streams_confined
streams_confined --> off
# /usr/sbin/setsebool streams_confined true
```

Confined and vetted PEs

When defining security policies, it is good practice to give the minimum necessary permissions to processes. PEs come in two distinct types:

- ▶ Some PEs do not access external resources; they consume streams from other PEs and produce streams for other PEs only.
- ▶ Some PEs need to access external resources; these PEs include sources, Sinks, and Operators integrating with external systems, such as databases or external analytics algorithms.

When running in confined mode, it is possible to differentiate between these two types, run them in two separate security domains, and define separate security policy modules for each one.

If a PE needs access to external systems, you can vet the code to ensure that it is appropriate and conforms to security standards. If so, then you can label the PEs as vetted. This PE will now run in a separate security domain (`streams_vetted_t`) than the unvetted PEs (which run in `streams_confined_t`).

You can define SELinux policy modules to permit access from the PEs in each domain to the external resources. When running in this dual-domain mode, it is recommended that the `streams_confined_t` be kept with minimal access while `streams_vetted_t` is configured with permissions to allow access to non-Streams systems.

4.8 Failover, availability, and recovery

Streams provides a number of facilities to allow recovery from failed hardware and software components. Support for the following failure recovery situations is provided:

- ▶ Recovery of Operator state.
- ▶ Recovery of failed Processing Elements.
- ▶ Recovery of failed application hosts (hosts running only the Host controller and PEs).
- ▶ Recovery of management hosts (a Host running any of the management services, that is, SRM, SAM, SCH, AAS, or NSR).

These recovery situations are discussed in the following sections.

4.8.1 Recovering Operator state: Checkpointing

As discussed in 4.4.2, “Overcoming performance bottleneck Operators” on page 140, stream computing works on data in motion, but some Operators do allow state to be maintained and windows of historical data to be managed. If the state of an Operator is not persistent, restarting the application for routine purposes or after failure will potentially cause a gap in your analysis.

The SPADE compiler provides a mechanism to checkpoint the state of an Operator at regular intervals; this compiler can be enabled by specifying a checkpoint directive in the Streams Application code. In Example 4-26, the checkpoint directive causes the ProcessedText Operator to checkpoint its state to the file system every 10 seconds.

Example 4-26 Checkpoint: Restartable and mobile directives

```
stream RawText(Text: String)
  := Source()["ctcpns://myRawTextAddress/"]{}
  -> restartable=true

stream ProcessedText (sourceID: String, TextMetaData: String)
  := Udoop (RawText)
    ["ProcessUnstructuredText"]
    {}
  -> checkpoint=10, mobile=true # checkpointed every 10 seconds
```

Note: When you checkpoint a UDOP or UBOP, the compiler will generate template code that will need to be populated to save the state of the Operator. You do not need to write any additional code to checkpoint the Streams built-in Operators.

Operators can be labeled with directives to indicate whether they are restartable and mobile or not. Any module that is checkpointed is automatically considered to be restartable.

In Example 4-26 on page 175, the RawText Operator has been labeled with a restartable=true directive, so this Operator will be automatically restarted on the same Host by the SAM service if it should fail. The ProcessedText Operator has been labeled with a checkpoint and mobile=true directive, so this Operator will be automatically restarted, potentially on a different Host by the SAM service if it should fail.

These directives are considered by the Streams compiler when fusing Operators together into PEs. Operators with incompatible directives will not be fused into the same PE.

In addition to the checkpointing facility provided by Streams, it is possible to develop an Operator to explicitly persist its state and read it if it is restarted.

4.8.2 Recovering Processing Elements

The deployment status of each PE is maintained by the Streams management services for the Instance. Should the Processing Element or its Host fail, then the status of the PE will be indicated as “TERMINATED” with reasons “CRASH” or “FAILURE”.

Example 4-27 shows the use of the **streamtool lspes** command to list the status of Processing Elements for an Instance. A number of PEs are running (PE IDs 0 and 1 are shown) and the PE with ID 7 has terminated. The RC is the reason code and F indicates Failure.

Example 4-27 Listing the status of Processing Elements

```
$streamtool lspes -i inst
```

PeId	State	RC	LC	Host	JobId	PeName
0	RUNNING	-	1	redbookstreams	0	BN.PE_BIOP_AverageNetwor...
1	RUNNING	-	1	redbookstreams	0	BN.PE_BIOP_BadgeEvents.2
...						
7	TERMINATED	F	1	redbookstreams	0	BN.PE_BIOP_NetworkEvents...
...						

PEs can be defined as restartable or mobile, based on the properties of the Operators in the PE (see 4.8.1, “Recovering Operator state: Checkpointing” on page 175):

- ▶ Processing Elements marked as restartable will be automatically restarted when they fail or crash.
- ▶ Processing Elements marked as restarted and mobile will be automatically restarted on different hosts when they fail or crash.
- ▶ Processing Elements not marked as restartable can only be rerun by cancelling and resubmitting the application as a new job.

Note: When a Processing Entity or a Streams job is restarted, entries in the data streams may be missed.

4.8.3 Recovering application hosts

If an application hosts fails, then the SRM service detects that the Host Controller on the Host is no longer running. As a result, the statuses of the PEs that were running on the Host failure are marked as Unknown.

The hosts' status can be shown by running the **streamtool getresourcestate** command and the PE status can be shown by running the **streamtool lspes** command, as shown in Example 4-28. In this example, the RBCompute3 Host has failed and the PEs running on this Host are now in an unknown state.

Example 4-28 Host status and Processing Element status when the Host fails

```
$ streamtool getresourcestate -i clusterinst
Instance: clusterinst@streamsadmin Started: yes State: READY Hosts: 5
(1/1 Management, 3/4 Application)
```

Host	Status	Schedulable	Services
RBmanage	READY	-	READY:aas,sam,sch,srm
RBCompute1	READY	yes	READY:hc
RBCompute2	READY	yes	READY:hc
RBCompute3	DOWN	no(hc-failed)	DOWN:hc
RBCompute4	READY	yes	READY:hc

```
$ streamtool lspes -i clusterhost
```

PeId	State	RC	LC	Host	JobId	PeName
0	RUNNING	-	1	RBCompute1	0	BN.PE_BIOP_AverageNet...
1	Unknown(RUNNING)	-	1	RBCompute3	0	BN.PE_BIOP_BadgeEvents.2
2	RUNNING	-	1	RBCompute2	0	BN.PE_BIOP_BadgeEvents...

There are a number of ways that an administrator can address an application Host failure:

- ▶ It is possible to restart the Host controller service on the original Host by running the **streamtool restarthost** command. If the command is successful, then the status of the PEs running on the Host can be determined and they will be labeled as running or terminated as appropriate.
- ▶ It is possible to quiesce the Host controller service on the failed Host by running the **streamtool quiescehost** command. The Host will still be part of the Instance, but it will not be used to run Processing Entities. PEs that were executing on this Host can then be restarted on other hosts.
- ▶ It is possible to remove the Host from the Instance by running the **streamtool rmhost** command or to remove the Host controller service from the Host by running the **streamtool rmhost** command. PEs that were executing on this Host will be marked as Terminated and can then be restarted on other hosts.

The SAM service will move restartable and mobile PEs to other hosts if possible. If SAM cannot move the PEs to other hosts, the PEs will remain in the TERMINATED state.

4.8.4 Recovering management hosts

It is possible to store the state of the critical management services and recover their state when the services are restarted. To accomplish this goal, you need to configure a DB2 database into which the runtime data of the Streams services is stored. The steps necessary to configure the DB2 database are detailed in the *Streams Administration Guide*.

Set the RecoveryMode property for the Streams Instance to get the management services to save their states to the recovery database. The RecoveryMode can be set when creating the Instance or can be set in an existing Instance; both commands are shown in Example 4-29.

Example 4-29 Setting the RecoveryMode while creating an Instance or for an existing Instance

```
$ streamtool mkinstance -i inst --numhosts=1 -property RecoveryMode=on  
or..  
$ streamtool setproperty -i inst RecoveryMode=on
```

Note: You can use the **streamtool setproperty** command to set the RecoveryMode for a started Instance, but the change will not take effect until the Instance is restarted.

After the RecoveryMode has been set to on and the Instance has been started, the Streams management services will store their runtime states to the configured DB2 database.

Note: SAM, AAS, and SRM are the only services that store their state to database; the SCH and SWS services are stateless and support recovery without needing a database. The NSR service does not support recovery via the database.

If the Host where a management services is now located fails, then when the service is restarted on another Host, it will restore the latest state from the recovery database.

For example, suppose that, for example, the SAM services should fail. An administrator user can restart the SAM service and this will recover the Application and Processing Entity state from before the server failure. You can show the status of the Streams services for an Instance by running the **streamtool getresourcestate** command, as shown in Example 4-30. This example shows a failed SAM service on the RBManage Host.

Example 4-30 Showing a failed SAM service

```
$ streamtool getresourcestate -i clusterinst
Instance: clusterinst@streamsadmin Started: yes State: READY Hosts: 5
(0/1 Management, 4/4 Application)
```

Host	Status	Schedulable	Services
RBmanage	SOME_DOWN	-	DOWN:sam READY:aas,sch,srm
RBCompute1	READY	yes	READY:hc
RBCompute2	READY	yes	READY:hc
RBCompute3	READY	yes	READY:hc
RBCompute4	READY	yes	READY:hc

You can restart the SAM service by running the **streamtool restartservice** command, as shown in Example 4-31.

Example 4-31 Command to restart the SAM service

```
$ streamtool restartservice sam -i clusterinst
```

If the Host itself has failed, then you can use the **streamtool addservice** command to move the service to a new Host. Example 4-32 shows how to move the SAM service to the “RBCompute1” node.

Example 4-32 Command to move the SAM service to a new Host

```
streamtool addservice sam -i clusterinst --host RBCompute1
```

4.9 Environment profiling

The Streams environment implements a Performance Counter interface. This allows you to monitor the performance of the Streams runtime components and the Processing Elements that comprise your applications.

You can visualize the performance of the environment using the IBM InfoSphere Streams Studio development environment, with Performance Visualizer windows or, with streamtool commands.

In addition, if you compile an application in profiling mode, you can generate resource usage statistics that allow the compiler to intelligently fuse the Operators to improve performance.

Performance counters

The Performance Counter interface is implemented by Streams runtime components (services) and Streams Operators. This means that you can use the same mechanisms to monitor the entire Streams system.

The aim of the Performance Counter Application programming interface is to expose performance counters that provide a view of the performance-critical activities carried out by Streams components, including runtime services, PEs, and Operators.

Note: It is possible to define your own performance counters as part of the Performance Counter Interface; this is covered in the *Streams Programming Model and Language Reference Guide*.

***perf_client* command to display the performance counters**

The **perf_client** command can be used to display the performance counters for a Streams Instance or to save the performance counters to file at particular intervals. Example 4-33 shows the use of the **perf_client** command to show the performance counters for PE number 32.

Example 4-33 Showing performance counters for a PE using the perf_client command

```
$ streamtool runcmd -i inst perf_client -l --pe=32
txtuples: 114874
rxtuples: 115414
txbytes: 3194514
rxbytes: 10168984
```

Example 4-34 shows the use of the --count and --loop options to repeatedly show the performance counters for a Processing Element at regular intervals.

Example 4-34 Showing performance counters five times every 4 seconds

```
$ streamtool runcmd -i inst perf_client -L --pe=32 --count=5 --loop=4
txtuples: 27714, rxtuples: 28256, txbytes: 770590, rxbytes: 2489470
txtuples: 47659, rxtuples: 48199, txbytes: 1325457, rxbytes: 4246623
txtuples: 74018, rxtuples: 74559, txbytes: 2058339, rxbytes: 6569143
txtuples: 93459, rxtuples: 93999, txbytes: 2599199, rxbytes: 8282083
txtuples: 113366, rxtuples: 113907, txbytes: 3152587, rxbytes: 10036228
```

Example 4-35 shows a way to run the **perf_client** command for multiple Processing Entities.

Example 4-35 Showing performance counters for multiple PEs

```
$ for pe_no in 1 3 10 2 8 9 11 14; do echo "PE: $pe_no" `streamtool
runcmd -i spade perf_client -rL -p $pe_no`; done
```

```
PE: 1 txtuples: 20324, rxtuples: 0, txbytes: 955356, rxbytes: 0
PE: 3 txtuples: 120, rxtuples: 0, txbytes: 5260, rxbytes: 0
PE: 10 txtuples: 346242, rxtuples: 0, txbytes: 30506952, rxbytes: 0
PE: 2 txtuples: 0, rxtuples: 10162, txbytes: 0, rxbytes: 477678
PE: 8 txtuples: 0, rxtuples: 82, txbytes: 0, rxbytes: 6907
PE: 9 txtuples: 0, rxtuples: 115414, txbytes: 0, rxbytes: 10168984
PE: 11 txtuples: 0, rxtuples: 5, txbytes: 0, rxbytes: 328
PE: 14 txtuples: 0, rxtuples: 2, txbytes: 0, rxbytes: 168
```

The **perf_client** command can also be used to monitor the Streams services and log details to a file. The full options for **perf_client** can be seen by running **streamtool runcmd perf_client -help**; these options are explained in more detail in the *Streams Installation and Administration Guide*.

Streams Studio monitoring facilities

The Streams Studio development environment comes with views to allow you to display the performance counters. These views are the Streams Live Graph and Streams Live Graph Outline, as shown in Figure 4-15.

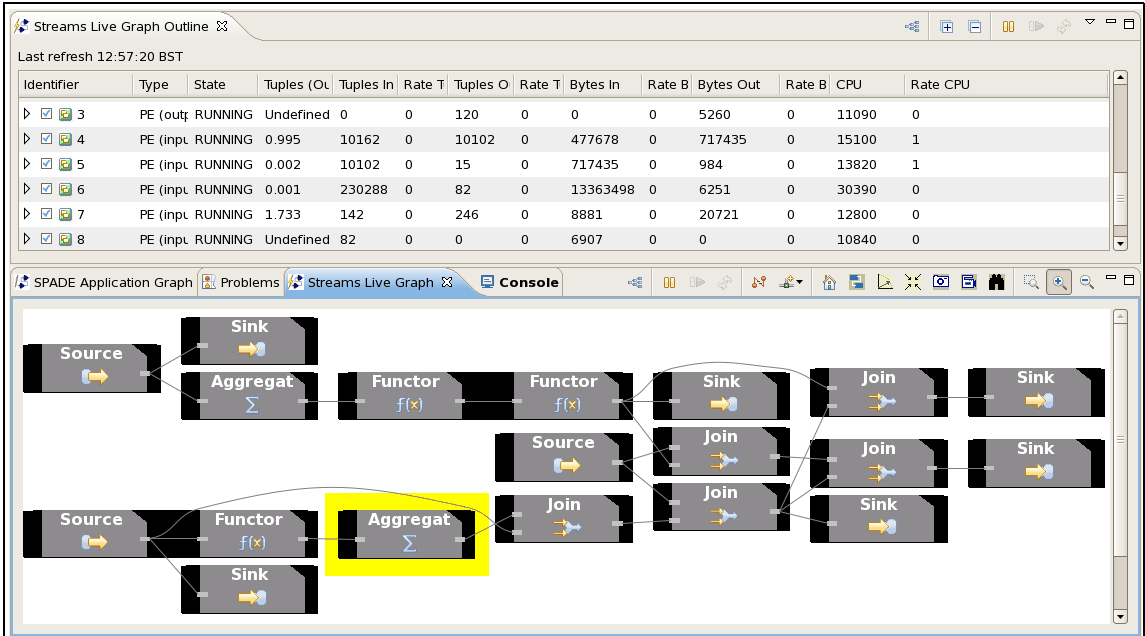


Figure 4-15 Displaying performance counters in the Streams Studio

From the Streams Live Graph or the Streams Live Graph Outline View, you can right-click a PE and select **Launch PerfViz for this PE** to open the performance visualizer window for the PE. The performance visualizer plots the input and output statistics for the PE in a graph that periodically updates. The performance visualizer for a Processing Element is shown in Figure 4-16.

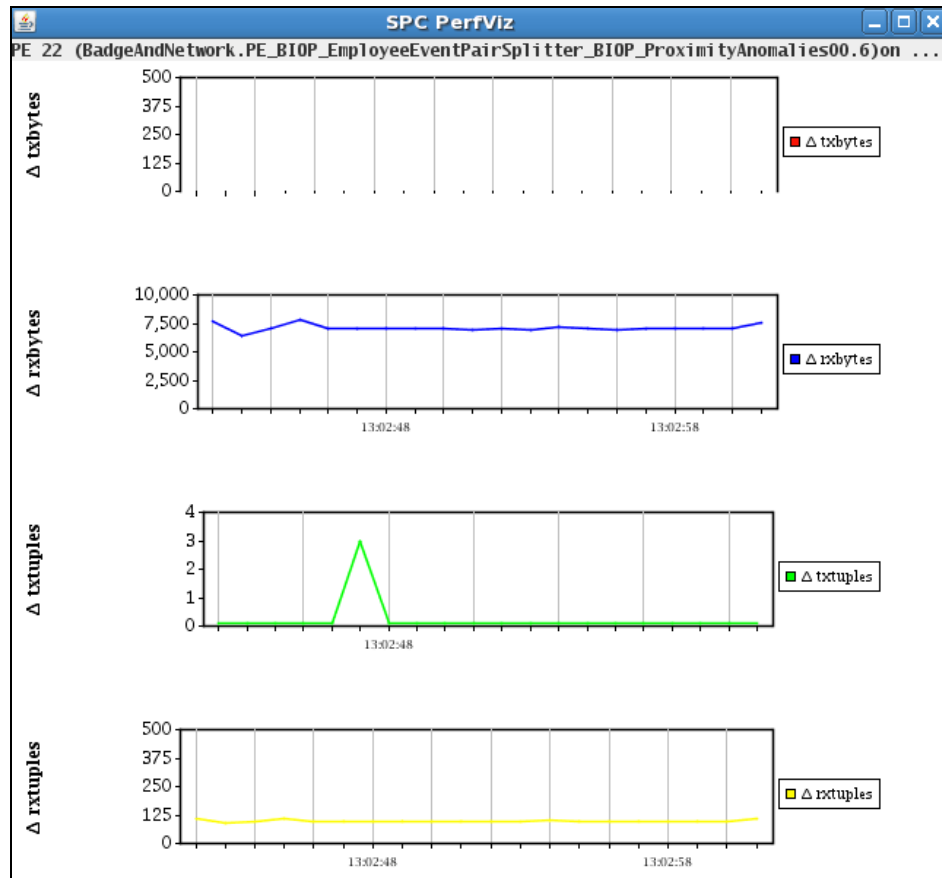


Figure 4-16 Performance visualizer window for a Processing Element

Profiling an application for compiler optimization

It is possible to profile an application while it runs to determine the resource usage of the individual Operators. This information can then be used by the optimizing compiler to determine how to fuse Operators in Processing Elements for best performance.

To profile an application, compile it with the -P switch. When you run this compiled application, the following profiling statistics are gathered:

- ▶ For each input and output port, the stream rate in tuples/second
- ▶ For each input and output port, the stream rate in bytes/second
- ▶ The fraction of the CPU utilized by each Operator

When the profiling application is cancelled, the profiling statistics are written to the file system where the application is compiled
(<applicationDirectory>/etc/stat/topology.<applicationName>).

You then use the spadestats.pl script to process these raw statistics and produce a summary of the statistics, as shown in Example 4-36. In this example, we process the profiling statistics from an application called “BN”. This command generated a file named <application directory>/etc/stat/topology.<application name>.stat.

Example 4-36 Summarizing profiling statistics

Run the following from the application source directory:
\$ spadestats.pl etc/topology.BN etc/stat

You can recompile your application by using the -p FINT option to specify intelligent fusing of Operators using the profile statistics. This is shown in Example 4-37; note the extra line generated by the compiler stating *Optimizing Operator graph partitioning...Success*; this is the additional activity of the intelligent compiler.

Example 4-37 Compiling an application with intelligent fusing of Operators

```
spadec -f BN.dps -p FINT
Using auto-managed instance 'spade', which is set as the default
instance in the environment
Compiling 'BadgeAndNetwork.dps'...
Creating Operator graph partitioning ...
Optimizing Operator graph partitioning...Success...
Creating application model ...
Reusing host list from an already-running instance 'spade'...
Creating stream types ...
Creating SPADE Operators ...
Creating Processing Elements (PEs) ...
Creating InfoSphere Streams artifacts ...
Creating makefile ...
Creating JDL and PE binaries ...
Building PE_BIOP....
....
Summary allocation:
```

redbookstreams : will host 17 PEs

Total number of jobs 1

Total number of PEs 17

Total number of PEs to be placed at compile time 17

Total number of hosts used for PE placement computed at compile time 1

Total number of hosts to be used 1

SPADE compilation has completed successfully



IBM InfoSphere Streams Processing Language

In the previous chapters, we described IBM InfoSphere Streams (Streams) positioning, concepts, process architecture, and examples of the types of problems that can be solved by Streams.

In this chapter, we describe the language that Streams Applications are developed in and provide code examples related to real-world scenarios.

More advanced features of the language, such as user-defined Operators, are covered in Chapter 6, “Advanced IBM InfoSphere Streams programming” on page 257.

5.1 Language elements

In this section, we describe the language elements that make up a Streams Application.

To describe the elements of the language, we introduce a simplified example in a mobile telecommunications company.

In this example, a mobile phone operating company is concerned about how their wireless infrastructure is performing. Customers have been complaining of calls dropping or being unable to make calls reliably. Consequently, the company is concerned that issues with cell broadcast masts may be impacting service revenues and customer satisfaction. Therefore, the company has built a Streams Application to monitor calls and perform call re-routing around defective masts.

Figure 5-1 shows an example Streams data flow. Note that it consists of a number of Operators, Sources, Sinks, and named flows (streams).

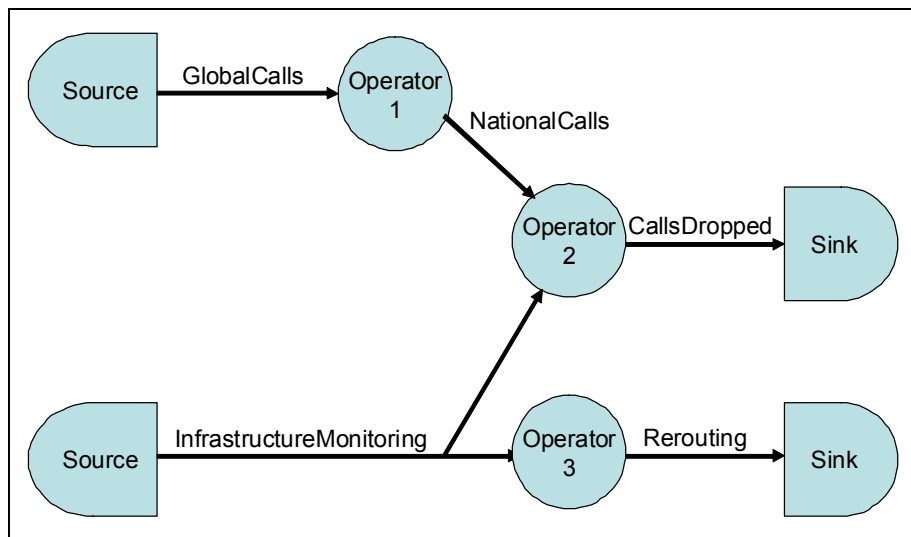


Figure 5-1 Example data flow for a mobile call re-routing application

The Source and Sinks shown in the diagram are used by a Streams Application to connect to outside data sources; they are the boundaries of the application as far as we are concerned. The Source Operator reads information from an external input and the Sink Operator writes the data to the external destination.

This Streams Application is designed to receive real-time data feeds that are read by the two Source Operators shown on the left side of Figure 5-1.

The input streams to the application provide:

1. Data on the calls being made by the company's customers in real-time (GlobalCalls)
2. Real-time data on the health of transmission masts (InfrastructureMonitoring)

We call the customer call data stream *GlobalCalls* because that stream contains call data for calls irrespective of whether their customer is making the call while in the company's home country or abroad.

Mast failure information however, is available only from the masts located in the company home country.

To avoid the processing impact on all the Tuples in the GlobalCalls stream further on in the flow, the Streams Application filters the GlobalCalls stream, removing any data about calls made on other countries' mast infrastructure. This filtering is performed in Operator 1.

It is not always possible to determine a call termination reason from the call Tuples alone; the customer may have ended the call normally or the call may have been ended due to being interrupted by a fault in, for example, the mast being used while the call is active. Operator 2 correlates the call data coming in through the National Call stream with the mast health monitoring information in the InfrastructureMonitoring stream to determine which calls were terminated specifically due to an infrastructure issue.

Operator 3 contains an internal re-routing table and, on the basis of failures, sends re-routing commands to the lower output Sink.

In Streams Processing Language, the flow described above is expressed at a high level, as shown in Example 5-1.

Example 5-1 Expressing this data flow in Streams Processing Language

```
stream GlobalCalls(...) := Source()[...]{...}
stream NationalCalls(...) := operator1(GlobalCalls)[...]{...}
stream InfrastructureMonitoring(...) := Source()[...]{...}
stream CallsDropped(...) :=
operator2(NationalCalls,InfrastructureMonitoring)[...]{...}
stream Rerouting(...) := operator3(InfrastructureMonitoring)[...]{...}
Null := Sink(CallsDropped)[...]{...}
Null := Sink(Rerouting)[...]{...}
```

Note that each stream is first defined on the left side of an assignment (an assignment is represented by := in Streams). Immediately following the := assignment is the Operator name responsible for producing that stream.

Also note that both the Source and Sinks are also considered Operators. Sources read external sources and convert the data from an external representation to the Streams representation. Similarly, Sinks do the reverse: The Streams internal format is converted to an external representation.

In Figure 5-1 on page 188, there is no stream shown beyond the Sink; what happens outside the Source and Sink is outside the scope of the Streams Processing Language.

The end of a flow is denoted with the word Null (instead of a named stream), as shown in the last line of Example 5-1 on page 189. (The term *Nil* may also be used instead of Null. Both Nil and Null mean the same thing; use whichever one you prefer to use.)

Note that within the parenthesis immediately following the Operator name is the list of one or more streams that are inputs to that Operator. For example, in the sample code shown, GlobalCalls is the input stream to Operator1.

Note also that the example code contains three types of braces: (), [], and {}.

The contents of the braces will be explained later in this chapter, but they might contain stream metadata, Operator input arguments, Operator parameters, and Operator output Attribute derivation expressions.

5.1.1 Structure of a Streams Processing Language program file

Streams Processing Language files (source files) are ASCII text files that commonly have a .dps extension. As text files, they can be written using any text editor of your choice, but most commonly are developed and maintained using IBM InfoSphere Streams Workbench.

Source files are compiled to binary code using the supplied Streams compiler.

Tip: If you prefer to use a text editor rather than the Workbench to edit your Streams source files, syntax highlighters for vi, JEdit, and GNU emacs text editors are provided with the software.

Each .dps file commonly expresses a flow from Source(s) to Sink(s) via one of more Operators connected by named streams.

The Streams Processing Language supports comments. Single line comments are prefixed by the # character. Anything after this character to the end of the line is ignored by the compiler. If comments are required to span multiple lines, the start of the comment section should be preceded by /* and ended with */.

Each source file is composed of up to six sections of which two are mandatory. The six sections, with square braces headings, are:

1. [Application]
2. [Typedefs]
3. [Libdefs]
4. [Nodepools]
5. [Program]
6. [FunctionDebug]

The [Application] and [Program] sections are mandatory, and are described together. The others sections are optional, and will be discussed subsequently.

► [Application] and [Program]

A simple .dps file containing the mandatory sections is shown in Example 5-2.

Example 5-2 A sample .dps file for the re-routing application

```
#Call Re-routing applllication
#
[Application]
    Routing info
[Program]
stream GlobalCalls(...) := Source()[...]{...}
stream NationalCalls(...) := operator1(GlobalCalls)[...]{...}
stream InfrastructureMonitoring(...) := Source()[...]{...}
stream CallsDropped(...) :=
operator2(NationalCalls,InfrastructureMonitoring)[...]{...}
stream Rerouting(...) := operator 3(InfrastructureMonitoring)[...]{...}
Null := Sink(CallsDropped)[...]{...}
Null := Sink(Rerouting)[...]{...}
```

Following the [Application] line is the name of the application (*Routing* in this example) followed by a keyword that indicates the debug/logging level required. The example shows that the information logging level has been selected (info). The alternatives in descending order of verbosity of logging are trace, debug, info, and error.

The logging level keyword can also be completely omitted, in which case no logging occurs.

Following the [Program] section heading is the body of the code. In this case, we have inserted the prior example (incomplete at this stage).

The optional sections are:

► [Typedefs]

Under this optional section header, the developer can create aliases for built-in data types to the types they intend to use in the program. In Example 5-3, the developer has aliased ListofMasts to the built-in type IntegerList. Using you own type names is commonly done for code readability, development standards, and type enforcement reasons.

Example 5-3 [Typedefs]

```
[Typedefs]
typedef ListofMasts IntegerList
```

► [Libdefs]

Under this optional section header, the developer can include references to header files that describe the interfaces to external libraries and the file-system paths to any library object code files. External library reference is often required if user-defined Operators are included in the solution. Examples of this heading are given in Chapter 6, “Advanced IBM InfoSphere Streams programming” on page 257.

► [Nodepools]

Under this optional section header, a pool of UNIX® hosts can be defined. While an application can be optimized by both the Streams compiler at compile time and at run time in the Streams deployment architecture, there may be criteria (for example, for development or production constraints) where it is beneficial to specifically control placement of Operators. This specific placement of Operators can be achieved using *nodepools*. We have included examples of this heading in Chapter 6, “Advanced IBM InfoSphere Streams programming” on page 257.

► [FunctionDebug]

Under this optional section header, developers may enter expressions for test purposes only. Placing expressions in this section causes them to be evaluated at compile time rather than at run time, as shown in Example 5-4.

Example 5-4 Using [FunctionDebug]

```
[FunctionDebug]
print timeMicroSeconds()
#
```

The above will cause the compiler to output:

Expression timeMicroseconds()evaluates to ‘123345621235’

Note that the number 123345621235 shown in Example 5-4 on page 192 will vary, as it corresponds to the actual compile time. It represents the number of microseconds since the epoch (January 1, 1970).

5.1.2 Streams data types

In this section, we describe the data types supported by Streams.

A stream has an associated schema, that is, the particular set of data types that it contains.

A schema consists of one or more Attributes. The type of each Attribute in the schema may be selected from one the following high level categories:

1. Basic data types
2. Lists and matrix data types (repeating combinations of basic data types)
3. Unstructured data

These categories are further described as follows:

1. Basic data types

The following basic data types are supported:

- Byte: 8-bit number
- Short: 16-bit number
- Integer: 32-bit number
- Long: 64-bit number
- Boolean: True or false
- String: A character sequence (up to 65,535 characters are allowed per string)
- Float: Single precision floating point (IEEE754)
- Double: Double precision floating point (IEEE 754)

Note that Strings are dynamically sized and created; it is not mandatory to specify a length of the String in the program code.

2. Lists and Matrix data types

Streams provides built-in support of lists and matrixes. The lists and matrixes may contain of any of the basic data types. Matrices may contain any of the basic data types with the exception of Byte or String data types. Functions and arithmetic can be applied to either sections of, or the entire set of, data held in list or matrix variables directly in one line of code. This benefit extends to include matrix multiplication.

The following are the supported types of List data:

- ByteList: A list of bytes
- ShortList: A list of short integers
- IntegerList: A list of integers
- LongList: A list of longs
- DoubleList: A list of doubles
- BooleanList: A list of Booleans
- StringList: A list of strings

The following types may be used to represent matrixes of the basic types:

- ShortMatrix
- IntegerMatrix
- LongMatrix
- FloatMatrix
- DoubleMatrix
- BooleanMatrix

3. Unstructured data types

Streams supports reading unstructured data in a number of ways, including using the ability of the Source Operator to ingest binary payloads alongside structured data and also through User-defined Operators.

Reading unstructured data is described in Chapter 6, “Advanced IBM InfoSphere Streams programming” on page 257.

5.1.3 Stream schemas

The data types may be grouped into a schema. The *schema* of a stream is the name that is given to the definition of the structure of data that is flowing through that stream.

For our simplified call routing application, the GlobalCalls schema contains the following Attributes:

- ▶ callerNumber: Long, #the customer number
- ▶ callerCountry: Short, #the country code where the customer is located
- ▶ startTimestampGMT: String, #call start time

- ▶ endTimeStampGMT: String, #call end time
- ▶ mastIDs: IntegerList #list of mast(s) used to make the call

Note that mastIDs is an IntegerList Attribute. In our example, it is assumed that each mast in the network has a unique numeric identifier. The mastIDs Attribute contains a list of the mast identifiers associated with the set of masts used to process that customer's call.

Within the Streams Application code, the schema of a stream can be referred to in several ways:

1. The schema may be explicitly listed out in full.

The schema can be fully defined within the Operator output stream parameters, as shown in Example 5-5, which shows the GlobalCalls schema.

Example 5-5 Schema fully defined in Operator output stream

```
stream GlobalCalls(callerNumber: Integer, callerCountry: Short,
startTimeStampGMT: String, endTimeStampGMT: String, mastIDs:
Integerlist) := Source(...)[...]{...}

stream NationalCalls(...) := ...
..
```

2. The schema may reference a previously defined vstream.

The schema definition can be created ahead of the Operator definitions using the keyword *vstream*, which stands for virtual stream. The keyword *vstream* allows the Streams developer to associate a name with that schema and subsequently refer to that name within the code rather than the entire definition in full each time. This reference to the user defined name uses the *schemaFor* keyword in the code body, as shown in Example 5-6.

Example 5-6 Using the vstream and schemaFor keywords

```
vstream callDataTuples(callerNumber: Integer, callerCountry: Short,
startTimeStampGMT: String, endTimeStampGMT: String, mastIDs:
Integerlist)
..
..
..
stream GlobalCalls(schemaFor(callDataTuples) := Source(...)[...]{...})
```

In the above example, the schema is given the name *callDataTuples*, which is then referred to in the definition of the GlobalCalls. Note that it is also possible to define a schema by referencing a previously defined Stream (which may be defined a vstream, or another, concretely-defined stream).

3. The schema may be defined using a combination of the previously described methods.

In this last variant, a schema for a stream may be defined by a combination of methods, as shown in Example 5-7. In this example, another Attribute is being added to the list of Attributes already present.

Example 5-7 Defining a stream schema using a combination of methods

```
vstream callDataTuples (  
    callerNumber : Long,  
    callerCountry : Short,  
    startTimestampGMT : String,  
    endTimestampGMT : String,  
    mastIDS : IntegerList  
)  
  
stream globalCalls (schemaFor(callDataTuples), anotherAttribute :  
Integer) := ...
```

Where required, streams of the same schema may be collected together on a single input port of any Operator.

5.1.4 Streams punctuation markers

Streams supports the concept of punctuation marks, which may be considered as markers inserted into a stream. These markers are designed so that they can never be misinterpreted by streams as Tuple data. Some streams Operators can be triggered to process groups of Tuples based on punctuation boundaries or they can insert new punctuation marks into their output stream(s).

5.1.5 Streams windows

In previous chapters, we have described the need for windows in the stream. Conceptually, we are working on streams of data that have no beginning or end. However, the Streams Operators processing may need to group sections of the data stream with similar Attributes or time intervals. The Streams Processing Language enables this type of grouping using a features called windows.

Streams can be consumed by Operators either on a Tuple-by-Tuple basis or through windows, which create logical groupings of Tuples.

windows are associated and defined on the inputs to particular Operators. If an Operator supports windows, each stream input to that Operator may have

window characteristics defined. Not all streams Operators support windows; this is dependant on the Operator functionality.

The characteristics of windows fall under the following headings:

1. The type of window
2. The window eviction policy
3. The window trigger policy
4. The effect the addition of the *perGroup* keyword has on the window

These characteristics are described in the following sections:

1. The type of window

Two window types are supported: Tumbling windows and Sliding windows.

- Tumbling windows

In Tumbling windows, after the trigger policy criteria is met, the Tuples in the window are collectively processed by the Operator and then the entire window contents are discarded, as shown in Figure 5-2.

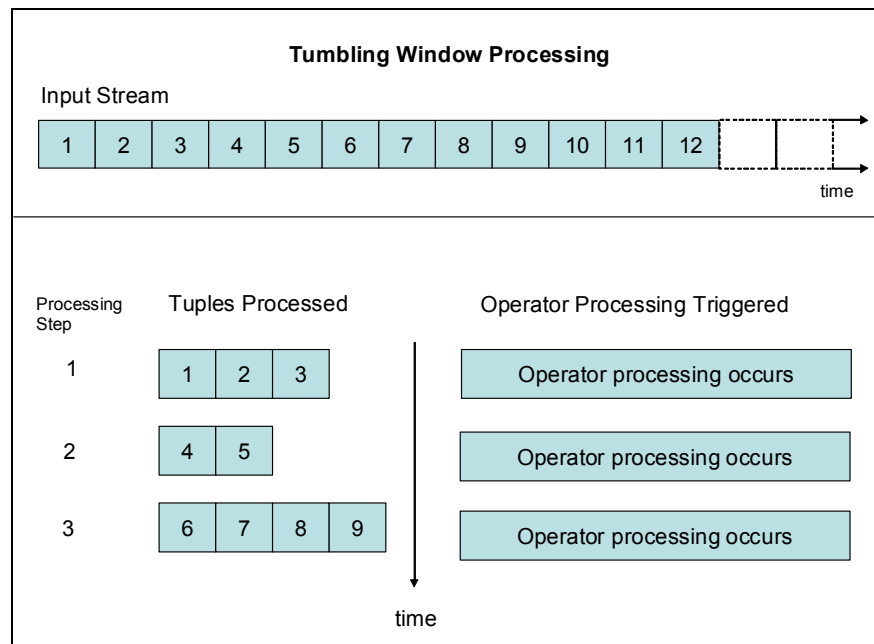


Figure 5-2 Tumbling windows

- Sliding windows

In Sliding windows, newly arrived Tuples cause Tuples to be evicted if the window is already full. In this case, the same Tuple(s) can be present for multiple processing steps, as shown in Figure 5-3. In contrast to Tumbling windows, with Sliding windows, any given Tuple may be processed by an Operator more than once.

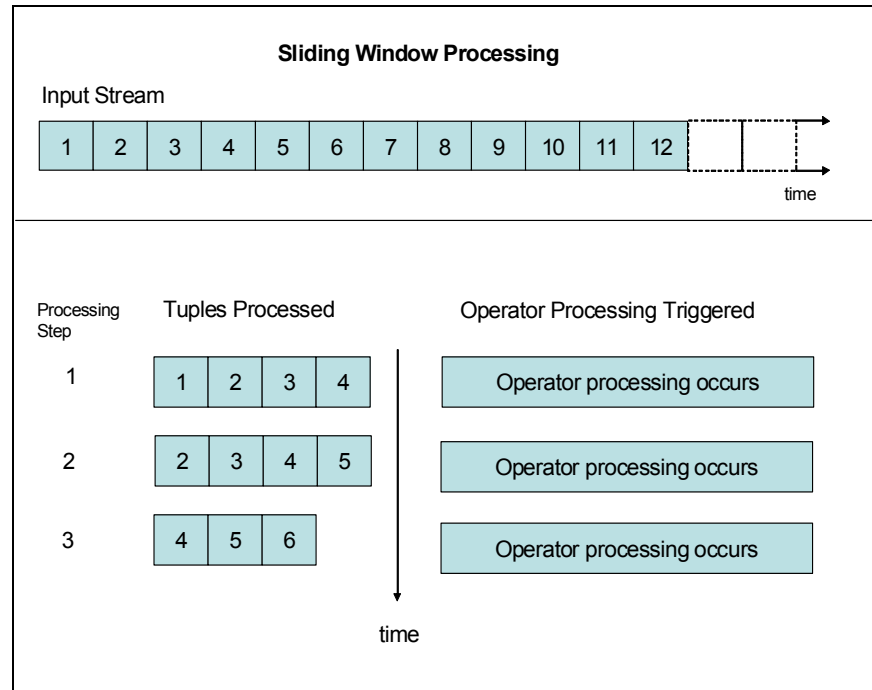


Figure 5-3 Sliding windows

- The window eviction policy

The eviction policy is what defines when Tuples are removed from the window. The eviction policy may be specified, after a selected numeric Attribute increases by an amount beyond a threshold, by using the following criteria:

- Quantity of Tuple
- Age of Tuples
- Punctuation marks (special Tuple separators understood by Streams)

- The window trigger policy

The window trigger policy defines when the Operator associated with the window performs its function on the incoming data, for example, a period of time or a number of Tuples.

4. The effect the addition of the perGroup keyword has on the window

For supported Operators, the use of the optional perGroup keyword will cause multiple window Instances to be created for the stream in question. The number of Instances will depend on the number of distinct values for the selected Attributes, that is, the runtime values of the Attributes used to define the group, as shown in Figure 5-4.

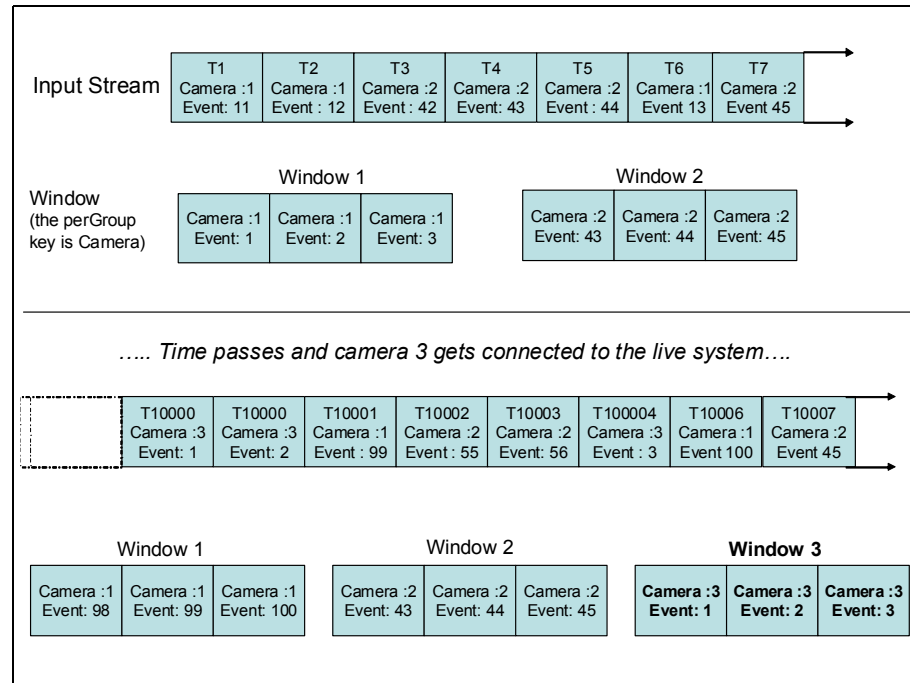


Figure 5-4 Use of perGroup keyword

As the application runs and new combinations of Attribute key values are received that have not so far been encountered, additional windows Instance are created dynamically. Each window Instance on the stream will operate as described above independently of each other within the common set of characteristics.

Referring to Figure 5-4, Streams is being used as part of a security monitoring solution. The system has motion sensitive security cameras that send captured image events when a security event triggers them. Sliding windows have been defined that have an eviction policy of three Tuples. Therefore, the latest three events will be present in each window. The `perGroup` keyword has been used and the group key is the camera ID, which means that the number of physical cameras does not need to be hardcoded anywhere in the application. Rather, the

application will dynamically create additional windows as cameras are added and start sending security event data.

Language syntax for windows

For Operators that support windows, and where windows are required by the developer, the language requires the definition of the window characteristics be between < ' and ' > characters. The window definition is placed after the relevant stream name and in the *input* to the relevant Operator, as shown in Example 5-8.

Example 5-8 Defining a group of Sliding windows

```
vstream someSchema(...)
stream myStream(schemaFor(someSchema))
    := myOperator(inputStream1<count(20),count(1), perGroup>)
    [group definition parameters]
    {...}
```

In the above example, each Sliding window will have an eviction policy count(20). This means that when the 21st Tuple arrives, the oldest Tuple (in order of arrival) will be evicted. The trigger policy has been specified by count(1). This means the Operator performs its function on the set of rows in the window every time a new Tuple arrives and after any evictions that its arrival may cause.

A set of such Sliding windows is created on the Tuples in inputStream1 because the perGroup keyword was used. The number of windows in the set will correspond to the group definition parameters supplied by the developer between the square braces, that is, in the Operator parameters section. These parameters need to identify the set of Attributes used to define a group.

In Example 5-9, we defined a single Tumbling window on the input port to the Operator. The Tumbling window triggers the Operator action every 60 seconds on whatever Tuples have arrived in that time. The Tuples in the window are then completely discarded, thus emptying the window. The time is then reset and the cycle repeats. There is only one window associated with inputStream1 because the perGroup keyword was *not* used.

Example 5-9 Defining a single Tumbling window

```
vstream someSchema(...)
stream myStream(schemaFor(someSchema))
    := myOperator(inputStream1<ttime(60)>)
    [...]
    {...}
```

For Tumbling windows, the trigger policy is explicitly specified (with or without the perGroup keyword). The options are the following:

- ▶ <punct()>: Punctuation-based Tumbling windows
- ▶ <count()>: Number of Tuples based Tumbling windows
- ▶ <time(seconds)>: Time-based Tumbling windows
- ▶ <attrib(attribute-delta)>: Change in a non-decreasing numeric amount

The eviction policy does not need to be explicitly defined for Tumbling windows. This is because after the Operator is triggered and completes its processing on the Tuples in the window, all of Tuples in the window are evicted (the eviction policy always matches the trigger policy).

For Sliding windows, both the eviction and trigger policy must be specified and they must be specified in that order (again, with or without the perGroup keyword). There are three choices for each policy:

- ▶ <count()>: The number of Tuples based Sliding windows
- ▶ <time(seconds)>: The time-based Sliding windows
- ▶ <attrib(attribute-delta)>: The change in a non-decreasing numeric amount

Excluding perGroup, this results in a choice of nine possibilities of Sliding window definition, specified in <evictionPolicy,triggerPolicy> order:

1. <count(n),count(n)>
2. <count(n),time(seconds)>
3. <count(n), attrib(attribute,delta)>
4. <time(seconds),count(n)>
5. <time(seconds),time(seconds)>
6. <time(seconds,attrib(attribute,delta)>
7. <attrib(attribute,delta),count(n)>
8. <attrib(attribute,delta),time(seconds)>
9. <attrib(attribute,delta),attrib(attribute,delta)>

Sliding windows do not support a trigger or eviction policy based on punctuation.

Example 5-10 illustrates a Sliding window using an Attribute-based eviction policy and a time-based trigger policy. In this example, temperature data is collected from national weather monitoring stations. Each weather observation has a weather observation time. The observations are guaranteed to arrive in order of observation time and observations arrive approximately every minute from each station. However, it is not guaranteed that observations will not be missed, or that the number of seconds past the top of the minute will be the same for each station.

Example 5-10 Sliding window with Attribute-based eviction policy

```
#calculate a 6 hour running average based on temperature obserations

#assumed schema of temperatureStream
vstream temperatureStream (
monitoringStation: String,
temperature: Float,
obserationTime: Integer
)

#define SECONDS_IN_6_HOURS 21600

stream dailyAverageTemps(
monitoringStation: String,
dailyAverageTemp : Float
)
:= aggregate(
temperatureStream<attrib(observationTime,SECONDS_IN_6_HOURS),time(90),
perGroup>
[monitoringStation]
{
monitoringStation := monitoringStation,
dailyAverageTemp := Avg(temperature)
}
)
```

In this example, we used the following preprocessor directive:

```
#define SECONDS_IN_24_HOURS 21600
```

This directive allows us to use the given meaningful name instead of just the integer constant anywhere in the code that follows. This configuration makes the code more readable. This and other features of the preprocessor are described in more detail in 5.3, “The Preprocessor” on page 226.

Note that the Sliding window uses an Attribute-based eviction policy and a time-based trigger policy.

The time-based policy is specified here as `time(90)`, so the aggregator outputs a Tuple containing the running average every 90 seconds.

Here is the Attribute-based eviction policy that was used:

```
attrib(observationTime,SECONDS_IN_6_HOURS)
```

The Attribute `observationTime` contains a number representing the time stamp of the observation. The number is the number of seconds since the epoch.

The selected eviction policy states that a Tuple will be evicted from the window when a new Tuple arrives that has an observation time more than six hours newer than it does.

The aggregator that specifies the `perGroup` grouping Attribute is `monitoringStation`, it uses the syntax `[monitoringStation]`, and calculates the average using the `Avg()` function.

5.1.6 Stream bundles

The language supports the definition and manipulation of bundles of existing streams. A bundle is the combination of Tuples from one or more streams with the same schema. Note that the compiler will check that the schemas are all the same, and that the application will not compile if the schemas are different.

The order of Tuples within a bundle are guaranteed to be the same as the order in their originating stream. Note however that the relative position of Tuples in a bundle originating from *different* streams cannot be guaranteed. Bundles are declared using the `bundle` keyword.

One or more additional streams may be added to an existing bundle after it has been previously declared and initialized.

Example 5-11 provides a code sample for defining bundles.

Example 5-11 Defining bundles

```
#bundle declared and initialized to empty stream
bundle myBundle1 := {}
```

```
#bundle declared and initialized to a combination of
#previously defined streams
bundle myBundle2 := {myInputStream, myOtherInputStream}
```

```
#previously defined stream added into previously defined bundle
myBundle1 += {inputStream3}
```

```
#the line below adds three more streams into myBundle1
#following which the bundle will contain tuples from
#four input streams
myBundle1 += {inputStream4,inputStream5,inputStream6}
```

A bundle may then be used by Streams Processing Language just as it would use any other stream.

The collection of Tuples comprising the originating streams making up the bundle may be accessed individually. This is achieved by using a range of index numbers, where the first stream to be added to a bundle has an index 0.

Referring to Example 5-11 on page 203, the expression `myBundle[1:3]` may be used to refer to the Tuples from three streams in the bundle having stream indexes 1, 2, and 3, which represents all the Tuples from `inputStream4`, `inputStream5`, and `inputStream6`.

`myBundle1[0:0]` may be used to refer to the Tuples that originated from the first stream (`inputStream3`).

Finally the index used to access the bundle may itself be a variable. So, for example, `myBundle[i:j]` refers to the collection of Tuples in the bundle originating from streams in the bundle with indexes *i* through *j*.

5.2 Streams Processing Language Operators

In this section, we describe the main features of the Streams Processing Language Operators, including code samples for each Operator.

This section is not intended to be a substitute for the language reference manuals included with the product software, but is intended as guide for the reader to more readily understand the examples shown in this chapter.

Which Operator(s) are needed

There are 10 built-in Operators that offer a range of powerful functions:

- ▶ Source: Reads external data into streams.
- ▶ Sink: Writes Streams data to external systems.
- ▶ Functor: Filters, transforms, and performs functions on the data.
- ▶ Sort: Sorts Streams data on defined keys.
- ▶ Split: Splits Streams data into multiple streams.
- ▶ Join: Joins Streams data on defined keys.

- Aggregate: Aggregates Streams data on defined keys.
- Barrier: Combines and coordinates Streams data.
- Delay: Delays a stream data flow.
- Puncctor: Identifies groups of data that should be processed together.

Table 5-1 is provided for users new to the language to use as a guide in selecting which Operator(s) to use for a given type of task.

Table 5-1 A high level guide for when to use the various Operators

Task	Operator(s) to use	Notes
Read or write external sources and target data.	<i>Source</i> to read and <i>Sink</i> to write (User-defined for more complex situations).	Source and Sink can connect to TCP or UDP sockets or read and write files.
Aggregate data.	Aggregate.	Window enabled Operator.
Join or lookup.	Join.	Window enabled Operator.
Pivot groups of data into lists.	Aggregate.	Window enabled.
Derive new or existing Attributes.	Functor.	Can retain values between Tuples, use state variables, and access prior Tuple data (history).
Filter data out.	Functor.	
Drop Attributes.	Functor.	
Split data with different characteristics into separate streams.	Split.	
Run multiple Instances of an Operator on different sets or partitions of data in the same stream for performance gains.	Precede that Operator with a split and have several copies of your required Operator in the job. Split the input stream on the data characteristics by which you want to partition.	

Task	Operator(s) to use	Notes
Collect data streams together.	If the streams have the same column Attributes, any input to an Operator can do perform this task via a semicolon separated list. If you need to coordinate the timing of data collection, use a Barrier or Delay Operator.	
Coordinate the timing of data in different streams.	Evaluate the use of windows, with the delay and the Barrier Operators corresponding to the distribution, synchronization, and downstream processing requirements.	
Sort data to meet downstream required dependencies.	Sort.	Sort is a window enabled Operator.
Drop one or more columns	All Operators can do this, but do not define a matching output Attribute for the input Attribute to be dropped.	

5.2.1 Operator usage language syntax

Note, as shown in Figure 5-5, that each input port can receive Tuples from multiple streams providing that when this done each of those streams has the same schema. However, each separate input and output port can have different schemas as required.

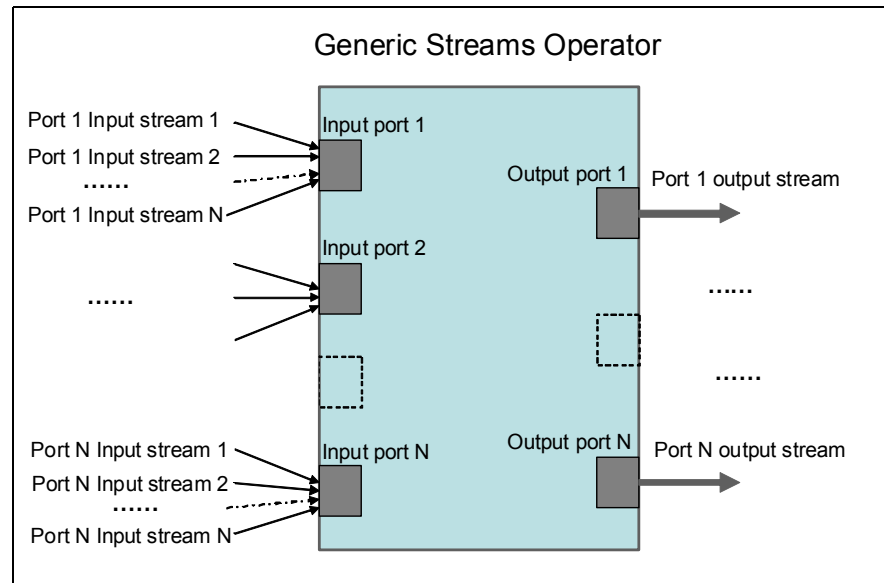


Figure 5-5 Operator inputs/outputs

The language syntax for all Operators follows the structure shown in Example 5-12.

Example 5-12 Operator language syntax

```
stream outputStream1name(outputStream1Schema)
stream outputStream2name(outputStream2Schema)
....
stream output StreamNname(outputStreamNSchema)
    := operatorName(
        inputPort1Stream1,inputPort1Stream2,..,inputPort1StreamM;
        ,...,...;
        ,...,...;
        inputPortNStream1,inputPortPStream2,..,inputPortPStreamM
    )
    [operatorParameters]
    {outputStream(s)Attribute(s)Assignments}
```

Note that the example shows the general case of an Operator having several input and output streams.

We have also shown several of those input streams going to the same input port on that Operator. Be aware that when this is done that the Streams on the same port must have the same schema.

Not all built-in Operators support multiple input ports or multiple output ports. The minimum is one input port or one output port, as in the case of a Source or Sink.

When Operators support more than one input or output port, in many cases the number of ports used for input or for output for any given Operator Instance is determined at compile time and corresponds to the required application.

The use of each Operator must include all the three types of parenthesis: (), [], and {}. The contents of the square braces [] contain optional Operator parameters, and the curly braces {} optionally contain derivations of the outputStreams Attributes. If the derivation for an output Attribute is not explicitly listed, the Attribute derivation is automatically matched to a corresponding input named Attribute.

A detailed description of the use of each of the Operators is provided in the following sections.

5.2.2 The Source Operator

The Source Operator may be used to read external data into a Streams Application from files and TCP/IP and UDP sockets. The file or socket is specified using a uniform resource identifier (URI). The Source Operator has one input port and one output port.

Using the Source Operator to read Tuples from a file

An example of the use of the Source Operator is shown in Example 5-13.

Example 5-13 Using the Source Operator to read from a file

```
stream myStream(callerNumber: Integer, name: String) :=  
Source()["file:///myFile.csv", csvFormat, noDelays] {}
```

Where:

- ▶ The file specifier is being used as part of the URI, which means the Tuples will be read from a file.
- ▶ The schema of the output stream is Integer, String, that is, the output Tuples will have two Attributes each.

- ▶ The URI file uses three / characters. In this case, the Streams program expects to find the file in a data directory beneath the directory of the .dps file containing the code, so in this example the relative file path would be ./data/myFile.csv. If you want to use an absolute file path, four / characters are required in the URI.
 - ▶ The noDelays modifier is specified so that the Source Operator does not expect a delay duration Attribute in the input stream.
 - ▶ The Operator parameters include the csvFormat parameter, indicating that the developer expects the data for each Tuple to be on a separate line with each Attribute comma separated in order of the schema definition.
- Omitting the csvFormat parameter would cause the Operator to expect each incoming Attribute to be denoted in comma separated name:value pairs with each Tuple new line delimited, as shown in Example 5-14.

Example 5-14 Example ASCII file containing two Attributes per Tuple in the default format

```
callerNumber:445558444123,name:Priti
callerNumber:15553120213,name:Davis
callerNumber:555551239871,name:Lee
```

Attributes that are of one of the supported List types are represented using square braces []. For example, [1,2,3] could be used for an IntegerList.

Boolean true or false Attribute values are represented in a file by the characters T and F, respectively.

Using the Source Operator to read from a socket

The Source Operator can read a stream of data from a TCP/IP socket, and also from a network socket. In this case, the corresponding data source may be a client or a server source, and in each case TCP or UDP variants are supported. The use of an external service by reference to a name rather than the traditional hostname:portname may also be used. However, this requires that a name lookup server be available in the network that can perform the translation of a service name to a hostname:portname.

In Example 5-15, the Source Operator is being asked to connect to the Host adapter with network address myexternalhost.ibm.com. The connection will attempt to be made over port 4512.

Example 5-15 Reading from a TCP/IP socket

```
stream mySourceStream(<schema>) :=
    Source()["stcp://myexternalhost.ibm.com:4512/"]{}
```

Note the use of stcp as the choice of protocol. Table 5-2 shows the supported options for the choice of protocol.

Table 5-2 Supported protocol specifiers

Protocol Specifier	Description
stcp	The Host (Source) adaptor is a TCP server.
ctcp	The Host (Source) adaptor is a TCP client.
sudp	The Host (Source) adaptor is a UDP server.
stcpns	As for stcp, except that the Host is specified using a name to be retrieved from a name server.
ctcpns	As for ctcp, except that the Host is specified using a name to be retrieved from a name server.
sudpns	As for sudp, except that the Host is specified using a name to be retrieved from a name server.

Using any of the last three entries in the table for a protocol specifier requires the presence of a service name lookup server in the network. In these cases, you must specify a service by name rather than hostname:portnumber in the URI, for example, ctcpns://stockquotes.

The Source Operator and Tuple inter-arrival timing

The Source Operator by default adds an additional first Attribute to the output stream, which is the *inter-arrival delay* measured in seconds between the Tuples. This inter-arrival delay is frequently not required, because where timing is important, the Tuples may contain a time stamp Attribute. If inter-arrival delays are not required to be automatically added by the Source Operator, the parameter (or Operator modifier) ,nodelays should be specified in the Operator parameters section following the Source URI. This parameter is shown in most of the examples in this chapter.

The Source Operator and initialization timing delay

For testing purposes, where sample input data is collected from a file rather than from a continuous input stream, the Streams Application flow as a whole must be fully initialized prior to the Source Operator sending the first Tuple. This task prevents the first few Tuples from being processed and discarded. In this case, the optional ,initDelay=<value> parameter can be added to the Source Operator parameters section after the specified URI. The term <value> is a numeric constant representing the number of seconds to wait before sending the first Tuple.

The Source Operator and punctuation marks

Punctuation marks were introduced earlier in this chapter. A variation of the Source Operator definition can also insert punctuation marks into its output stream. This is done by specifying the trigger condition for inserting punctuation within the definition in the braces of the Source(<punctuation_specification>). In previous examples of the usage of the Source Operator, the contents of these particular braces have been empty.

The three variations for inserting punctuation marks are as follows:

- ▶ <punct()>: Insert a punctuation mark just prior to the end of the stream.
- ▶ <time(t)>: Insert a punctuation mark every t seconds.
- ▶ <count(n)>: Insert a punctuation mark every n Tuples.

Example 5-16 shows the three types of usage.

Example 5-16 The three variations of Source Operator that introduce punctuation

```
vstream myStream(i:Integer, j:String)

#variation 1: introduce a punctuation mark as an end of stream marker
#ie. when the input stream is exhausted:
stream myStreamofTuples(schemaFor(myStream))
    := Source(<punct()>)
    []
    {}

#variation 2: insert punctuation every 60 seconds
stream myStreamofTuples(schemaFor(myStream))
    := Source(<time(60)>)
    []
    {}

#variation 3: insert punctuation every 100 Tuples
stream myStreamofTuples(schemaFor(myStream))
    := Source(<count(100)>)
    []
    {}
```

5.2.3 The Sink Operator

The Sink Operator is used to perform a function that is the reverse of the Source Operator function, that is, it receives an input stream and externalizes the Tuples to the required output format, which may be transmitted to a TCP/IP socket, or it writes the Tuple data to a csvformat file on the UNIX file system.

Accordingly, the Sink Operator has one input port and no output ports.

The Sink Operator can use the same protocol specifiers for the URI as the Source Operator by adding the *perfcounter:///* literal format URI. In the case of the perfcounter, the Streams performance Counter API is used to create one performance counter per numeric Attribute in the stream schema. The performance counters can be accessed using *perfviz*, which is the IBM InfoSphere Streams general-purpose performance counter GUI-based client.

Note also that this is the only Operator that uses a Null or Nil statement.

Punctuation and the Sink Operator

By default, the Sink Operator writes punctuation markers to text-based output, for example, they are marked by the # character. To disable this behavior, use the dropPunct modifier.

5.2.4 The Functor Operator

The Functor Operator is used where the developer needs to perform functional transformations, including filtering and dropping Attributes from the input stream.

The Functor has one input port and one output port. It processes the input on a Tuple by Tuple basis, so window functionality is not required or supported.

A Functor Operator is declared as shown in Example 5-17.

Example 5-17 Declaring a Functor Operator

```
stream <stream-name> (<stream-schema>)
    := Functor (<input-streams>)
    < #optional user defined logic section enclosed in pair of <>
      # ** declare and initialise user defined variables here **
    >
    <
      # ** user defined custom logic goes here**
    >
    [ <filter-condition>, dropPayload? ]
    {
<output-attribute-expression1>,...,<output-attribute-expression_n> }
```

Note the function has optional sections which, if provided, are enclosed within two pairs of angle braces (< and >). User-defined variables declared as part of Functor Operators are prefixed with a \$ to distinguish them from schema Attributes. The user-defined logic may make use of both declared user-defined variables and schema Attributes, and may include conditional statements (if, elseif, else, and also while loops).

In Example 5-18, we show code that calculates a running total that is output at every Tuple and reset upon a change of the value of the key Attribute.

Example 5-18 Using Functor logic to derive a new output Attribute

```
vstream mySchema(key: integer, data: integer)

stream myStreamWithTotal(schemaFor(mySchema), total: integer)
    := Functor(myInputStream)
< # variable declaration and initialisation
    Long $total := 0;
    Integer $lastkey := -1;
>
< # custom logic
    if ($lastkey != key) {
        $total = data
        lastkey = key
    } else {
        $total = $total + data
    }
>
[true] # filter condition - we use true, so all tuples get through
#
#now assign the output attributes, note the use
#of the $total variable.
{ key := key, data := data, total := $total }
```

Functors can also easily access previous input Tuples, which is referred to as *history access*. For example, to achieve access to the previous tenth Tuple Attribute, called temperature, you can use the expression `^10.temperature`.

Knowing this, Example 5-18 can be simplified, eliminating the need for the lastkey variable. The custom logic section could now read as follows:

```
< # custom logic
    if (key != ^1.key) {
        $total = data
    } else {
        $total = $total + data
    }
>
```

Where the `^1.key` refers to the value of the previous Tuple key Attribute.

Note the following points regarding Functor history processing:

- ▶ The language currently supports a literal number in the code (known at compile time) that defines the number of Tuples to go back. A variable number is not supported. The number to go back has no limit, but be aware that the compiler may allocate buffering to achieve a limit.
- ▶ If code refers to Tuple numbers prior to the first Tuple received by the program, then the Attributes will have default values, which are 0 for numerics, an empty string for strings, false for Booleans, and an empty list for lists.

The filter condition can include the dropPayload keyword. Binary payload data was introduced in the description of the Source Operator.

Including the dropPayload keyword in a Functor will cause the Attribute containing that payload data to be dropped (not propagated beyond the Functor).

Punctuation and the Functor Operator

The Functor Operator preserves any punctuation it receives. Punctuation therefore passes through the Functor Operator unchanged in order, on receipt, and irrelevant of any Functor processing or filtering.

If you must insert punctuation, based on Tuple Attribute input condition(s), the Puncture Operator should be used.

5.2.5 The Aggregate Operator

The Aggregate Operator is able to perform summarization of user-defined groups of data where the summarization is based on data of all values, or may be divided into groups where one or more Attributes have the same values. This Operator has one input port, one output port, and supports both Sliding and Tumbling windows.

The Aggregate Operator offers the functions shown in Table 5-3 (for a group of Tuples).

Table 5-3 *Aggregator functions*

Function	Description
Cnt()	Count the Tuples in the group.
Min()	Determine the minimum value of an Attribute.
Max()	Determine the maximum value of an Attribute.
Avg()	Determine the average of an Attribute.

Function	Description
Sum()	Accumulate the total of an Attribute.
First()	The first value of an Attribute.
Last()	The last value of an Attribute.
DCnt()	The count of distinct values of the Attribute.
Col()	Create a Streams List data type containing all the values of an Attribute.
DCol()	As for Col(), except that the list contains only distinct values.
Mcnt()	Total number of Tuples in windows of all groups.
Gcnt()	Number of groups.
Vcnt()	Create a Streams List data type containing the group sizes for all groups.
Bcnt()	Create a Streams List data type containing the count of each distinct Attribute value in the group.

Example 5-19 shows an example use of the Aggregate Operator.

Example 5-19 Example Aggregate Operator usage

```
#Assumed aggregator input stream schema
vstream stockPrices (stockSymbol: String, price: Float)

#output stream schema
vstream avgStockPrices (stockSymbol: String, avgPrice:Float)

stream avgStockPrice(schemaFor(avgStockPrices)
    := Aggregate(inputStockStream<count(30),count(1), perGroup>
    [stockSymbol]
    {stockSymbol := stockSymbol, avgPrice := Avg(price)})
```

Where:

- ▶ We are using a Sliding window that includes the perGroup keyword. The window has an eviction policy that is count based at 30 Tuples and the trigger policy is also count based for each Tuple. The Attribute being used for the group key is stockSymbol.
- ▶ We are using the Aggregate Operator Avg() function, which calculates averages. This means that we are calculating the moving average for each stock symbol. The example will output a calculation of the average of the last

30 Tuples received for each stock symbol. Such a calculation might be used in company trend analysis.

Punctuation and the Aggregate Operator

The prior text has noted that the Aggregate Operator may use punctuation as a method of trigger and eviction policy for Tumbling windows (but not Sliding windows).

The Aggregate Operator inserts a window marking punctuation into the output stream after the aggregation operation is completed and a batch of Tuples are emitted.

5.2.6 The Split Operator

The Split Operator has one input port and may have one or more output ports. It may be used to split an input stream into various output streams. Which stream(s) an input Tuple is dispatched to is dependent on the parameters specified for the Operator. These streams typically will route a Tuple based on Attribute characteristics.

The Split Operator does not need to support windows because it functions on each input Tuple one at a time, routing it to the required output port(s).

The input and output stream schemas must be the same for the Split Operator.

The Split Operator is declared as follows:

```
stream OutputStream1(OutputStreamSchema)
stream OutputStream2(OutputStreamSchema)
..
stream OutputStreamN(OutputStreamSchema)
:= split (InputStream1,InputStream2..,InputStreamN)
[splitParameters]
{}
```

There are three mutually exclusive forms of splitParameters:

- ▶ An expression returning an Integer
- ▶ An expression returning an IntegerList
- ▶ A key-mapping file

These forms are described in more detail in the following text. In each of the descriptions, it is assumed that the Split Operator has N output ports numbered consecutively 1 through N. The expressions comprising splitParameters typically call for runtime comparisons with values of specific Tuple Attributes.

► An expression returning an Integer

In this form, a Tuple can only either be dropped or sent to exactly one output port. If the integer expression evaluates to -1, the input Tuple is discarded. If the expression evaluates to 0, the Tuple is dispatched to output port 0 (the first port). If the expression evaluates to 1, the Tuple is dispatched to output port 1 and so on. In general, a non-negative expression value of e means the Tuple should be dispatched to output port $(e \bmod N)$, where N is the number of output ports. Example 5-20 shows use of the SplitOperator with a integer expression.

Example 5-20 Using the Split Operator with an integer expression split parameter

```
vstream businessAccounts(accountID: String, averageBalance: Float)
```

```
stream highValueAccounts(schemaFor(businessAccounts))
stream mediumValueAccounts(schemaFor(businessAccounts))
stream lowValueAccounts(schemaFor(businessAccounts))
:= split(inputStreamOfAllAccounts)
[select ( averageBalance >= 10000,0 select (averageBalance >=
5000,1,2))]
{}
```

High value businessAccounts are classified with average balances greater than or equal to 10000, medium value accounts are greater than or equal to 5000 and less than 10000, and low value accounts have average balances below 5000.

Note that we use the select conditional expression. The format is as follows:

```
select(<booleanExpression>, resultIfTrue, resultIfFalse)
for example if you wrote
i := select (true, 1,2)
after this line i would contain the number 1
whereas
i := select(false, 1,2)
after this line i would contain 2.
```

Note that in the example we nest the select calls to achieve the required result.

► An expression returning an IntegerList

In this case, the Tuple can be dropped or sent to one or more output ports simultaneously. In this form of split parameter, a list of integers is provided using a stream's IntegerList type. Each Integer in the list is then interpreted by the Operator in the same way as for the integer expression case previously shown. The benefit of this option over Integer Operator is that a Tuple can be

sent to several output ports simultaneously. Example 5-21 shows a split using an IntegerList.

Example 5-21 A split using an IntegerList

```
#in this example we split book authors into streams of the types of
#genre of books they have written
#authors often write books of more than one genre..

#schema of allAuthors stream:
vstream authorGenres(authorID: String: genresWritten: IntegerList)

stream SciFi(schemaFor(authorGenres))
stream Horror(schemaFor(authorGenres))
stream Crime(schemaFor(authorGenres))
stream Romance(schemaFor(authorGenres))
stream Fantasy(schemaFor(authorGenres))
:= Split(allAuthors)
[genresWritten]
{}
```

In Example 5-21, the input stream *allAuthors* contains Tuples that contain author identifiers and a list of the genre(s) of book(s) they have written.

For example, if an input Tuple genresWritten Attribute contained the values [0,4], this would be used to indicate that the associated author has written books of both the SciFi and Fantasy genres, so this input Tuple would be dispatched to two of the five output streams.

► A key-mapping file

In this form, the Operator parameters contains a URI file and an Attribute to split. The URI file points to the key-mapping file.

The key-mapping file is in the .csv format. Each line contains a record of the following format:

<attribute-value>, <comma seperated list of ports>

If the input Tuple Attribute value matches the first value on any of the lines, the Tuple is dispatched to the list of output ports given on the remainder of the line.

The benefit over the previous example is that the key-mapping file is an external configuration data file that could be changed or updated without requiring a code change. Example 5-22 provides an example using a key-mapping file.

Example 5-22 The Split Operator using a key-mapping file

```
#this is the same example as in Example 5-21 on page 218, that has been
re-written #to use a key-mapping file
#so the IntegerList is now not part of
#the input or output stream schema..
#in this example we split book authors into streams of the types of
#genre of books they have written
#authors often write books of more than one genre..

#schema of allAuthors stream:
vstream authors(authorID: String)

stream SciFi(schemaFor(authors))
stream Horror(schemaFor(authors))
stream Crime(schemaFor(authors))
stream Romance(schemaFor(authors))
stream Fantasy(schemaFor(authors))
:= Split("file:///mapping.txt", authorID)
[genresWritten]
{}

#the contents of the mapping.txt might contain
#1,0,3,4
#2,3
#3,1,2
#this would mean
#author 1 has written SciFi, Romance and Fantasy
#author 2 has written Romance
#author 3 has written Horror and Crime
```

Punctuation and the Split Operator

Similar to the Functor Operator, the Split Operator passes punctuation through unchanged. If the split has multiple output ports, an incoming punctuation is passed to all output ports unchanged. The Split Operator itself does not use punctuation to determine split functionality.

5.2.7 The Punctor Operator

The Punctor Operator has one input port and one output port. window grouping functions are not supported or required because the Operator works on a single Tuple at a time.

The primary use of the Punctor Operator allows the solution designer to insert punctuation marks into a stream based on required conditions. Typically, this is done in preparation for downstream Operator behavior, where it is required for a Tumbling window trigger and eviction policy based on punctuation.

The Operator parameters sections supports a Boolean expression that, when evaluated to true, will cause a punctuation mark to be inserted in the output stream. The expression may refer to prior Tuples using the syntax for history processing described for the Functor Operator. A before or after keyword is also included to determine if the punctuation is inserted before or after the Tuple, causing the expression to evaluate to true respectively.

The output Attribute assignment section of the Operator definition may include Attribute derivation expressions as per the Functor Operator.

In Example 5-23, we demonstrate the use of the Punctor Operator.

Example 5-23 Using the Punctor Operator

```
#in this example we insert punctuation on a key change in
#the data stream. The punctual marker is inserted at the end of each
series of a particular key value before the next series of tuples
having the key attribute with a different value.
```

```
#input and output stream schema for this example
vstream exampleSchema(key: Integer, data: String)
```

```
stream punctuatedStream(schemaFor(exampleSchema))
  := Punctor(someInputStream)
  [key != ^1.key, before]
  {}
```

5.2.8 The Delay Operator

The Delay Operator has one input port and one output port. Its function is to delay a stream by a number of seconds specified by a double parameter. It does not use windows because it operates on a Tuple by Tuple basis. A Delay Operator is typically used when you must closely synchronize Tuples from

different streams and relative timing is important to downstream Operator processing. In Example 5-24, we demonstrate the use of the Delay Operator.

Example 5-24 Using the Delay Operator

```
#delay a stream by 1.2 seconds
stream delayedStream(schemaFor(someSchema))
    := Delay(inputStream)
    [1.2d]
    {}
```

Punctuation and the Delay Operator

The Delay Operator passes through any punctuation marks in the stream unchanged and after the specified delay, while maintaining the relative position of the punctuation marker against the surrounding Tuples.

5.2.9 The Barrier Operator

The Barrier Operator is used to combine (or merge) multiple input streams to one output stream where the input streams are logically related. It has two or more input ports and one output port. An output Tuple is emitted only when an input Tuple has been received by each and every input port. The output Tuple Attributes can be explicitly derived in the output Attributes section of the Operator definition from some or all of the input Tuples set of Attributes.

The Barrier Operator is not required to support windows because it operates on a Tuple by Tuple basis for each input stream.

A typical usage is where a stream has earlier been split into two or more streams because performance can be gained by processing each of the two streams together before subsequently recombining using the Barrier Operator. In this scenario, it might be important that the order and number of Tuples in the prior split stream be maintained. In other scenarios of merging or combining streams, the Join Operator may be more appropriate.

The Barrier Operator does not require any parameters in the parameters section. We demonstrate the use of the Barrier Operator in Example 5-25.

Example 5-25 Using the Barrier Operator

```
#schemas of the input streams to the barrier
vstream stream1Schema(i:integer, a:String)
vstream stream2Schema(i:integer, b:String)
vstream stream3Schema(i:integer, c:String)
```

```
#schema of the output stream from the barrier
vstream outputStream(i:integer,x:String,z:String)

#operator usage
stream outputStream(schemaFor(outputSchema))
    := barrier(stream1; stream2; stream3)
    [] #no parameters for Barrier.
    { i := $1.i, x :=a, z :=c } #output attribute assignments.
#
#note that where input streams contain identicailly named
#attributes e.g 'i' in the above example we explicity refer the
#the input stream number to derive the output attribute from to
#eliminate any ambiquity as to which input stream attribute of that
#name we are referring to.
```

Punctuation and the Barrier Operator

The Barrier Operator does not propagate punctuation, because although incoming streams may have punctuation, the resulting output stream is an interleaved combination of the input Tuples and therefore it would not be possible to relate punctuation to the relevant stream.

5.2.10 The Join Operator

The Join Operator is used to perform a relational join of two input streams. A relational join may be inner, left, right, or full outer join. The join condition and the type of join are specified in the Operator parameters section.

In the description of the join functionality that follows, it is essential that the reader is familiar with the description of window characteristics provided in 5.1.5, “Streams windows” on page 196.

The Join Operator has two input ports and one output port. It supports Sliding windows but *not* the Tumbling windows for both its input ports.

The eviction policy for a join may be count, time, or Attribute-delta based, which is in line with normal Sliding window characteristics.

The trigger policy for a join is always count(1). A count(1) trigger policy means that the join processing always occurs as each input Tuple arrives on either input port. Because the trigger policy for each input port window is always count(1), Streams does not require it to be explicitly stated in the code of the Operator definition. Note that this may make the window definition in the code for the join initially appear to the reader to look like a Tumbling window definition, but remember that the join only supports Sliding windows.

The operation of the join is as follows. As each new Tuple is received on either port, the new Tuple is matched against all of the Tuples that are in the opposite port window. For each set of Tuples that satisfies the join condition, an output Tuple is constructed using the rules defined in the Operator Attribute derivation section of the code, which is between the curly braces.

The `perGroup` keyword may also optionally be used where required on either or both input streams. Note in this case that the time-based eviction policy is not supported with `perGroup` for the join. The choices are confined to either count-based or Attribute-delta based eviction. In this case, a number of windows are created on the relevant input streams grouped on the relevant distinct runtime values of the Attribute keys specified in the join criteria.

Example 5-26 demonstrates a left outer join.

Example 5-26 A simple Streams Application performing a left outer join

[Application]

joins deug

[Program]

```
stream decode(key: Integer, decode: String)
    := Source()
    ["file:///decode.dat", nodeLays, csvformat]
    {}

stream primary(key: Integer, someData: String)
    := Source()
    ["file:///primary.dat", nodeLays, csvformat, initdelay=5]
    {}

stream joined(key: Integer, decode: String, someData:String)
    := Join(primary<count(0)>; decode<count(1),perGroup>)
    [ LeftOuterJoin, {$1.key} = {$2.key} ] #join type and condition
    {$1.key, $2.decode, $1.someData} #output attribute derivations

Nil:=Sink(joined)["file:///joinedOut.dat"], nodeLays]{}

```

This example joins two .csv format input files on the integer key Attribute. The decode file is assumed to contain a text description associated with numeric key identifiers. A left outer join is used in case a key value cannot be decoded. In that case, the key is still output to the output file with a value of an empty string for decoding purposes.

The primary stream contains a list of keys to be decoded. Note that it flows into the first or left input port to the Join Operator. A <count(0)> eviction policy for this is used, meaning that the input Tuple is immediately discarded after processing.

The decode stream uses a <count(1),perGroup > window definition. This means that a number of windows are created in the stream on the second input port to the join, each of size one Tuple. The number of windows will equate to the number of distinct values for the key in the decode file. This approach means that we hold the decode values continuously in memory of the Operator.

Assume we have the Tuple data shown below for the primary.dat and decode.dat files:

```
primary.dat
1,def
3,ghi
0,abc
decode.dat
0,alpha
1,beta
```

The following result would occur in the decode.dat file:

```
1,beta,def
3,,ghi
0,alpha,abc
```

The second line of output contains an empty string for the decode Attribute as there is no matching key value for 3 in decode.dat.

Punctuation and the Join Operator

The Join Operator inserts punctuation in its output stream to delineate the sets of resulting matching Tuples for each pair of windows processed.

5.2.11 The Sort Operator

The Sort Operator sorts a set of Tuples in a window by a defined set of Attribute keys. Single and multiple sort keys are supported, each of which may be specified as *asc* or *desc* for ascending or descending sorts, respectively.

Sorts are typically used either as a requirement prior to exporting the Tuples via a Sink to meet an external target requirement, or for downstream Operator window requirements. For example, Attribute-delta based windowing requires the Tuples to be sorted on the relevant Attribute.

The Sort Operator has one input port and one output port and supports Sliding and Tumbling windows, with some restrictions.

Tumbling windows may be defined by count, time, or punctuation. Attribute-delta based windowing is not supported. After the Tumbling window is full, the sort is performed on the defined Attribute keys. Example 5-27 shows a Tumbling window-based sort.

Example 5-27 Tumbling window-based sorts

```
#schema of input and output streams
vstream streamSchema(key1:Integer, key2:Integer,someData:String)

stream sortedStream(schemaFor(streamSchema))
    := Sort(inputStream<time(60)>) #sort 60 seconds worth of
                                #tuples every 60 seconds
    [key1 (asc).key2 (asc)] #sort is ascending on key1,key2
    {}                    #auto assign output attributes here
```

In this example the Operator only ever processes the window when a Tuple is received. So, if a Tuple is not received in 60 seconds, the window will not be processed. The next Tuple to be received after 60 seconds will trigger the sort and ejection of Tuples. A consequence of this is that if we stop receiving Tuples, the last Tuples in the Sort Operator will never be output.

Sliding windows may be defined only by a count-based eviction policy. Other eviction policies are not supported for Sliding window-based sorts.

The trigger policy for sorting Sliding windows is always count(1). Rather than use count(1) explicitly, the key word *progressive* is used instead. The progressive key word is understood to have been chosen in this case to distinguish the window from a count based Tumbling window, and to avoid the developer attempting to use other currently unsupported trigger policies for the Sliding window.

Example 5-28 demonstrates the use of a Sliding window with a Sort Operator.

Example 5-28 Sliding window-based sort

```
schema of input and output streams
vstream streamSchema(key1:Integer, key2:Integer,someData:String)

#this example continuously sorts the last 100 tuples received and
outputs the tuples in descending order of key1,key2
stream sortedStream(schemaFor(streamSchema))
    := Sort(inputStream<count(100),progressive>)
    [key1 (desc).key2 (desc)]
    { key1:=key1, key2:=key2, someData:=someData }
```

In this example, 100 Tuples are read into the window. Then, as each new Tuple is read, it is placed in order in the window. Because the window is larger than the eviction policy size, the Tuple in the window with the highest values of key1, key2 is output.

Punctuation and the Sort Operator

The Sort Operator may process punctuation-based Tumbling windows as previously described.

Also, the Sort Operator inserts a window marking punctuation into the output stream following the completion of the sort processing on the group of Tuples in the window for all supported variations of windowing schemes described.

5.3 The Preprocessor

The IBM InfoSphere Streams Preprocessor makes a pre-pass through of the Streams Source code. It performs certain code conversions prior to full code compilation if certain preprocessor directives are included in the source code. The primary purpose is to assist the developer in code maintenance.

The following functionality is offered by Streams Preprocessor:

- ▶ #define substitutions
- ▶ #include files
- ▶ Application parameterization
- ▶ For loops
- ▶ Mixed mode processing with Perl

This functionality is described in the following text:

► **#define substitutions**

If you are familiar with the C or C++ programming languages, this description will be familiar. The `#define` function enables the developer to define literal textual replacements for a given keyword. The convention is to define the text to be replaced in all upper case. We demonstrate the use of this functionality in Example 5-29.

Example 5-29 Using #define

```
#define SECONDS_PER_DAY 86400
#define MY_MESSAGE "DAY CHANGE EVENT"
```

[Application]

ChangeOfDay #ouput a message on a change of day

[Program]

```
stream someOutputStream(someSchema)
    := Functor(someInputStream)
<
    String $message := MY_MESSAGE;
    boolean dayChange := false;
>
<

    if (mod(timeMicroSeconds())/1000000,SECONDS_PER_DAY) =0) {
        dayChange := true;
    }
    else {
        dayChange := false;
    }
>
[ dayChange ]
{ message := $message }
```

The code snippet in Example 5-29 uses the more meaningful text `SECONDS_PER_DAY` instead of 86400, and also defines a message to be output. The benefit is that it is easier to change the message text later on (for example, to another language or due to being referenced in several places) rather than searching through the code for all occurrences.

► **#include files**

The Streams Preprocessor has support for include files, also similar to the support provided by the C/C++ preprocessor.

The file naming convention is that Streams includes files that have the extension `.din` (rather than `.dps`).

The `#include` files are read literally by the preprocessor and placed into the `.dps` file, which includes them prior to passing the resulting code file to the Streams compiler proper.

Any valid streams code or preprocessor directives can be put in `#include` files (including references to further include files).

Typically, `#include` files can contain the following:

- Common code or preprocessor directives that need to be used in more than one Streams Source file. The benefit is that it can be maintained in one place.
- More complex logic, with the benefit that such logic can be more easily developed and tested in a separate file by a test harness prior to including it as part of the full application or to assist readability of the overall code structure.
- `vstreams` metadata definitions, with the benefit of also simplifying the repetition of the metadata of a stream referenced in several Source files.

We demonstrate the use of Streams `#include` files in Example 5-30.

Example 5-30 Using Streams #include files (part 1)

#following is contents of the source file `demoapp.dps`
[Application]
demoapp

[Program]
#include "vstreams.din"

#voStream is a vstream defined in `vstreams.din`
#it is referenced in several source files
stream outputStream(schemaFor(v0stream))
:= Source()
[]
{}

We demonstrate a second use of Streams `#include` files in Example 5-31.

Example 5-31 Using Streams #include files (part 2)

#following is contents of the source file `vstreams.din`

#streams metadata used in several streams and applications
vstream v0stream (

```

customerKey: Long,
customerName: String,
customerType: String
)

vstream vInputStream (
..
)

vstream ..

etc

```

Example 5-30 on page 228 and Example 5-31 on page 228 show examples of storing common vstream definitions in an #include file.

► Application parameterization

This Preprocessor feature allows the developer to pass parameters to the compiler, which are then in turn passed to determine the final shape of the code to be compiled. Within the code, %% may be used to determine the count of parameters.

There are two variants of the methods to pass parameters: by positional number and by parameter name.

For the positional number method, in general, %<N> may be used to reference parameter number N. For example, %1 would reference the first parameter, %2 would reference the second parameter, and so on.

We demonstrate referencing parameters in Example 5-32.

Example 5-32 Referencing parameters by positional number

```

#code snippet from file filter.dps
..
..
# filter stream by compiler parameter 1
stream filteredOutputStream(outputSchema)
  := Functor(inputStream)
    [ key="%1" ]
    {}

```

If the code in Example 5-32 on page 229 was compiled using `$ spacec -i filter.dps IBM`, then the line `key="%1"` would be translated to `key="IBM"` prior to the main compilation step.

In Example 5-33, we demonstrate passing parameters by name.

Example 5-33 Referencing parameters by name

```
#code snippet from file filter.dps
..
..
..
#define KEY
# filter stream by compiler parameter 1
stream filteredOutputStream(outputSchema)
    := Functor(inputStream)
    [ key=KEY ]
    {}
```

If this example was compiled using `$ spacec -i filter.dps KEY=3`, then the line `[key = KEY]` would become `[key=3]` prior to the main compilation step.

As you may note, the second form is similar to using a `#define`, but is passed as a compiler command-line parameter.

► Using for-loops

The Preprocessor can replicate sections of code using *for-loops* that define the section of code to be replicated. Note that this should not be confused with any type of runtime loop processing of your application. It is a Preprocessor loop and therefore generates code prior to the main compilation step.

Repeating sections of code may be required for:

- Analogous handling of different streams of data
- Splitting a stream and then performing identical parallel processing on each set of Tuples in each split flow for the benefit of better utilization of underlying parallel hardware

Preprocessor for-loops may be nested. The general form of a preprocess for a loop is:

```
for_begin @variable expression1 to expression2 step expression3
    loop_body # code to be replicated here..
for_end
```


The step expression3 is optional. If omitted, the step value will default to 1. The @variable is initially set to expression1 and then incremented by an amount specified by expression3 until it reaches or exceeds expression2.

In Example 5-34, we demonstrate the use of a Preprocessor for-loop.

Example 5-34 Using a Preprocessor for-loop with the Split Operator

```
vstream TradeQuote(...)
..
for_begin @S 1 to 5
    stream TradeQuoteStream@L(schemaFor(TradeQuote))
for_end
    := Split(TradeQuoteStream)
    [hashCode(symbol)]
    {}
```

The example will be converted to the following and then passed to the main compilation step:

```
stream TradeQuote_1(schemaFor(TradeQuote))
stream TradeQuote_2(schemaFor(TradeQuote))
stream TradeQuote_3(schemaFor(TradeQuote))
stream TradeQuote_4(schemaFor(TradeQuote))
stream TradeQuote_5(schemaFor(TradeQuote))
    := Split(TradeQuoteStream)
    [hashCode(symbol)]
    {}
```

The input stream has been split into five output streams.

The above technique is often used in Streams to partition the data, pass each partition into a separate Operator Instance, such as an aggregate or join, and then combine them again at some point for the benefit of parallel performance. In the above example, hash partitioning based on the stock symbol has been used. Subsequently combining the separate streams can be achieved with either a bundle or by using the Barrier Operators.

For-loops can be used to assist in the creation of trees or other types of replicated Operator graphs. The end condition for the loop may in turn be a Streams Application parameter, which can have the benefit of supporting the process of tuning a parallel Streams Application deployment onto different numbers of nodes in the cluster during testing and production. This is because different capabilities of hardware or data volumes may support different degrees of parallelism.

► Mixed mode processing with Perl

The described features of the Streams Preprocessor are suitable for many small to medium size Streams Applications.

However, for larger-scale applications, more extensive scripting support may be required. In this case, the language supports what are referred to as *mixed mode* programs. For mixed mode programs, the Preprocessor executes an additional step where sections of the code, which are written as Perl code, are used to generate Streams code.

Mixed mode programs need to have the extension `.dmm` rather than `.dps`. Streams uses Perl to convert `.dmm` files into `.dps` files before then passing the `.dps` file in turn through subsequent preprocessor and compilation steps as normal and as previously described.

Any Perl code used in a `.dmm` file must be enclosed in one of two following delimiters:

```
<% perl-code-here %>
```

In this form, the code between the delimiters is passed straight to Perl.

```
<%= perl_print_expression %>
```

This is a shorthand for a Perl print statement (that is, it is equivalent to it).

```
<% print print_expression; %>
```

Note that this shorthand saves the `print` keyword and includes the trailing semicolon.

We do not describe the features of the Perl in this book, as it is covered extensively in many other sources.

We also emphasize that this is a preprocessing step used to generate Streams code during compilation. Do not use the Perl code examples as a direct part of the run time of the application.

We demonstrate the use of Perl in a mixed mode source file in Example 5-35.

Example 5-35 A mixed mode Streams source file

```
#use perl to check the compiler command line arguments are
#correctly supplied
<%
use integer;
my $levels = $ARGV[0];
unless(defined($levels) and $levels>0) {
    print STDERR "Error - please specify degree of parallelism"
    exit 1;
}
%>
```

```
[Application]
    exampleMixedMode info
```

```

[Program]
vstream callDataRecords(numberCalled:String...)

stream callStream(schemaFor(callDataRecords))
    := Source(stcp(...))
    []
    {}

#use the split operator to partition the input stream
<%
for(my $i=1; $i<$levels; ++$i) {
%>
stream callDataRecordsPart_<%= "$(i)" %>(schemaFor(callDataRecords))
<% } %>
    := Split(callStream)
    [hashCode(numberCalled)]
    {}

..

```

In this example, the mixed mode application expects to receive a compile time parameter, which is the degree of parallelism into which the input call data records stream is split. The Perl code at the start performs a positive check that the parameter supplied exists, and is a positive integer.

We used the Preprocessor to control the number of output streams from the Split Operator.

If the code was compiled with `$ spacedc -i exampleMixedMode.dmm 3`, then the resulting Split Operator would appear as:

```

stream callDataRecordsPart_1(schemaFor(callDataRecords))
stream callDataRecordsPart_2(schemaFor(callDataRecords))
stream callDataRecordsPart_3(schemaFor(callDataRecords))
    := Split(callStream)
    [hashCode(numberCalled)]
    {}

```

5.4 Export and Import

In addition to making use of the Source and Sink Operators, Streams Applications can also communicate with each other using streams directly. This is achieved by exporting and importing streams.

Named streams can be imported or exported at any point in the flow.

Two types of export and import are supported.

- ▶ Point to point
- ▶ Publish and subscribe

These are detailed in the following sections.

5.4.1 Point to point

In point to point, one Streams Application exports a stream and a second Streams Application imports it.

An exported stream has the `export` keyword prefixed to the stream definition on the desired output of any Operator port.

Example 5-36 shows the output stream of a Source Operator being exported. Note that this stream may also optionally be consumed as normal by other Operators in the same application (as is the case in this example: The Sink Operator also consumes the stream in the same application that exports it).

Note that any given application can export multiple streams, but each stream must have a unique name within that application, which is normal for stand-alone applications.

To export a stream, just prefix the stream definition with the keyword *export*.

Example 5-36 A Streams Application exporting a stream (exporter.dps)

[Application]

exporter info

[Program]

```
export stream myExportedStream(key: Integer, data: String)
  := Source()
  ["file:///input.csv", nodeLays, csvformat, initdelay=5]
  {}
```

```
Null := Sink(myExportedStream)
      ["file:///output.csv", nodeLays, csvformat]
      {}
```

To import a stream, use the *import* keyword. Both the name of the application that is exporting the stream and the exported stream name from that application must be included as part of the imported stream definition. The referenced application and stream name are prefixed using the *tapping* keyword.

After defining the list of streams that are imported, those streams may be referenced on the input port of any consuming Operator(s) of the importing application, as with any conventional stream.

An imported stream is given a local stream name that may or may not be the same name as the name it had in the exported application.

Example 5-37 shows an application that imports the stream exported from the application shown in Example 5-36 on page 234. In this example, the imported stream with the name `myExportedStream` in the original exporting application is given the localname `myStream`.

Example 5-37 An application importing a stream (importer.dps)

```
[Application]
  importer info

[Program]
#imported stream definitions
import stream myStream (key: Integer, data: String) tapping
exporter.myExportedStream

Null := Sink(myStream)
      ["file:///imported.csv", nodelays, csvformat]
      {}
```

This example reads the imported stream and saves it to a `.csv` file. If you would want to run these examples, they may be compiled and run normally.

You may want to create a source file named `input.csv` in the program data directory that contains some test Tuples:

- 1, importing
- 2, and
- 3, exporting
- 4, with
- 5, streams

If the run is successful, you should see two newly created files in the data directory, `output.csv` and `imported.csv`, which both contain identical copies of the input data. This simple example demonstrates the similar usage of both internal and exported/imported streams.

Sequential input files are commonly used for testing, as in this case, but have the disadvantage that they are finite in length. The observant reader may have noticed that `exporter.dps` contained the Source Operator parameter `initdelay=5`.

This parameter causes the Source Operator to wait 5 seconds before sending the data through, which gives the `importer.dps` application time to fully initialize. If you launched the `importer.dps` application after 5 seconds, you would not see the exported file data.

You should also note that with an infinite source of Tuples (which might be typical in a Streams Application), the importing application could be launched at any time and it would obtain Tuples from the exporter only following the launch and initialization of the importer. If the exporting application also contained Operator code that consumed the streams (as in our example), those downstream Operators would not be kept waiting for the importing application to also start.

Conversely, an importing application will wait for Tuples forever if a corresponding exporting application of the required name and stream name is never started.

Finally, even though this method is called point to point, multiple importing applications can exist, each tapping the same exported stream.

5.4.2 Publish and subscribe

The publish and subscribe model extends the point to point model by adding the ability to associate named properties to a stream. An importing application can import streams from any application that exports streams, provided it has the properties named by the importer.

Therefore, the publish and subscribe model removes any need for the importing application to know both the name of the application exporting the stream and the exported stream name. Instead, an importing application can connect to an exported stream of any arbitrary name coming from any application exporting it, provided it has the specified named properties.

Within the *exporting* application, (the application *publishing* the stream), the `export properties` keyword pair is used together with a list of named properties that the designer is free to specify. The normal stream name and its deriving Operator definition then follows.

Within the *importing* application, the `import` and `tapping` keywords are used as before, but in this case, instead of using the `tapping application.stream` construct, as required by the point to point example, an XQuery expression is used to specify the required characteristics that the imported stream must have.

In most Streams Applications, simple XQuery expressions containing a list of required characteristic names can satisfy Streams Application designs that require the publish and subscribe model.

In Example 5-38, we have published two streams.

Example 5-38 Publishing two streams

```
[Application]
    publisher trace

[Program]
#include "schema.din"

export properties [category: "currentOperators"; type: "alloperators"]
    stream myStream(schemaFor(supplierSchema))
    := Source()
    ["file:///operators.csv", nodelays, csvformat, initdelay=10]
    {}

export properties [category: "currentOperators"; type "UKoperators"]
    stream ukSuppliers(schemaFor(supplierSchema))
    := Functor(myStream)
    [SupplierOrigin = "UK"]
    {}

#the below is included for debugging purposes to show the expected
#stream contents for those subscribers requesting access to our
#published stream with UKSuppliers attribute set
Null := Sink(ukSuppliers)
    ["file:///uksuppliers.csv", nodelays, csvformat]
    {}
```

This example includes the `schema.din` file, which contains the schema of the published streams:

```
#contents of schema.din:
vstream supplierSchema(supplierCode: Integer, supplierName: String,
SupplierOrigin: String)
```

Putting the published schema in an `#include` file means that the source code for subscribers can also include the schema easily.

The Source Operator reads the Tuple data used for this example from the `operators.csv` file. In our case, this file contains the following lines:

```
1,AT&T,USA
2,Vodafone, UK
3, Virgin,UK
4,T-Mobile, UK
5, Telefonica,Spain
6,Orange,France
7,Vivo,Brazil
```

5.5 Example Streams program

In this section, we describe a Streams Processing Language example that you could compile and run.

The infamous *Hello World* example program is widely accepted as originating from the book *The C Programming Language* by Kernighan and Ritchie. The scenario given is that your first program in a new programming language should do something simple, such as output a message that says “Hello World”. Performing this task means that the developer:

- ▶ Has gained access to a development environment
- ▶ Has accessed the compiler and has a properly installed development and runtime environment for the programming language concerned
- ▶ Knows how to use a compatible language editor
- ▶ Is able to compile and run the software

If the program successfully runs, the developer then knows that the proceeding tasks and environment are, at a basic level, ready to use for development.

In Example 5-39, the program reads from the `input.csv` file in the data directory and copies the Tuples it contains to the output file `output.csv`. The `input.csv` file is expected to contain Hello, World, and if the program runs correctly, this Tuple is copied to the output file `output.csv`. Relative file paths are specified, so the input and output files are expected to be in the data subdirectory of the program code directory.

Example 5-39 Hello World example

```
#read a file called input.csv containing the string Hello, World
#write it to output.csv
#
[Application]
```



```

        helloworld trace
[Program]
stream myStream(str1: String, str2: String)
    := Source()
    ["file:///input.csv",nodelays,csvformat]
    {}

Null := Sink(myStream)
    ["file:///output.csv",nodelays,csvformat]
    {}

```

5.6 Debugging a Streams Application

In this section, we describe some options and approaches for debugging a Streams Application. There are nearly as many methods to debug a Streams Application as there are business problems that you can solve with Streams. The sample Streams Application used in this section is listed in Example 5-40.

Example 5-40 Sample Streams Application used to demonstrate Streams debugging

```

[Application]
MyApplication_J1 debug

[Program]

stream InputFile (col1:Integer, col2:Integer)
    := Source()
    [
        "file:///InputFile.txt",
        nodelays,
        csvformat,
        throttledRate=1000
    ]
    {}

Null := Sink(InputFile)
    [
        "file:///OutputFile.txt",
        nodelays,
        csvformat
    ]

```

{}

Where:

- ▶ The example listed is a self-contained Streams Application, which means that it does not read from any imported Streams, it does not export any Streams itself, and it contains no references to externally defined user routines or functions.
- ▶ The Streams Application reads from one operating system file (`InputFile.txt`), and outputs two columns (`col1` and `col2`) in the Stream named `InputFile`.

This stream is consumed by the Sink Operator, which writes to an operating system file named `OutputFile.txt`.

In effect, this Streams Application is a simple file copy of every row and every column of `InputFile.txt` to `OutputFile.txt`.

- ▶ Not displayed are the contents of `InputFile.txt`. Per our example, this file contains two numeric (integer) columns separated by a comma delimiter, and a new-line character as the record separator.

This file contains 10,000 records, with values ranging from 10,000 to 20,000, increasing sequentially and by a value of one. The sample contents are shown below:

```
10000,10000
10001,10001
10002,10002
    {lines deleted}
19998,19998
19999,19999
20000,20000
    {end of file}
```

- ▶ Unique to this example are two modifiers of specific interest. These are the following:

- After the line naming this application (`MyApplication_J1`) is a modifier named *debug*.

This modifier to the Application Identifier outputs a given level of diagnostic information to a log file on the operating system hard disk, that is, the Streams system log file (or files), which has a variable name and location that can be configured.

Other values that may be placed in this location include `trace`, `(debug)`, `info`, `warn`, `error`, and `fatal`. Each value determines an increasing level of verbosity to the logging file contents.

- A modifier to the InputFile Operand named *throttledRate* will direct the output of records from the Stream named InputFile to output 1000 records, wait 1 second, then continue.

This modifier would likely be removed for any production Streams Application, but we find the ability to slow down the execution of a Streams Application to be useful when debugging.

5.6.1 Using Streams Studio to debug

The Streams Studio developers workbench has many visual components that enable effective debugging. In Figure 5-6 on page 242, we misspelled the name of the operating system file that serves as input for Example 5-40 on page 239. It is spelled `InputFile2.txt`, when the correct name is `InputFile.txt`. `InputFile2.txt` does not exist.

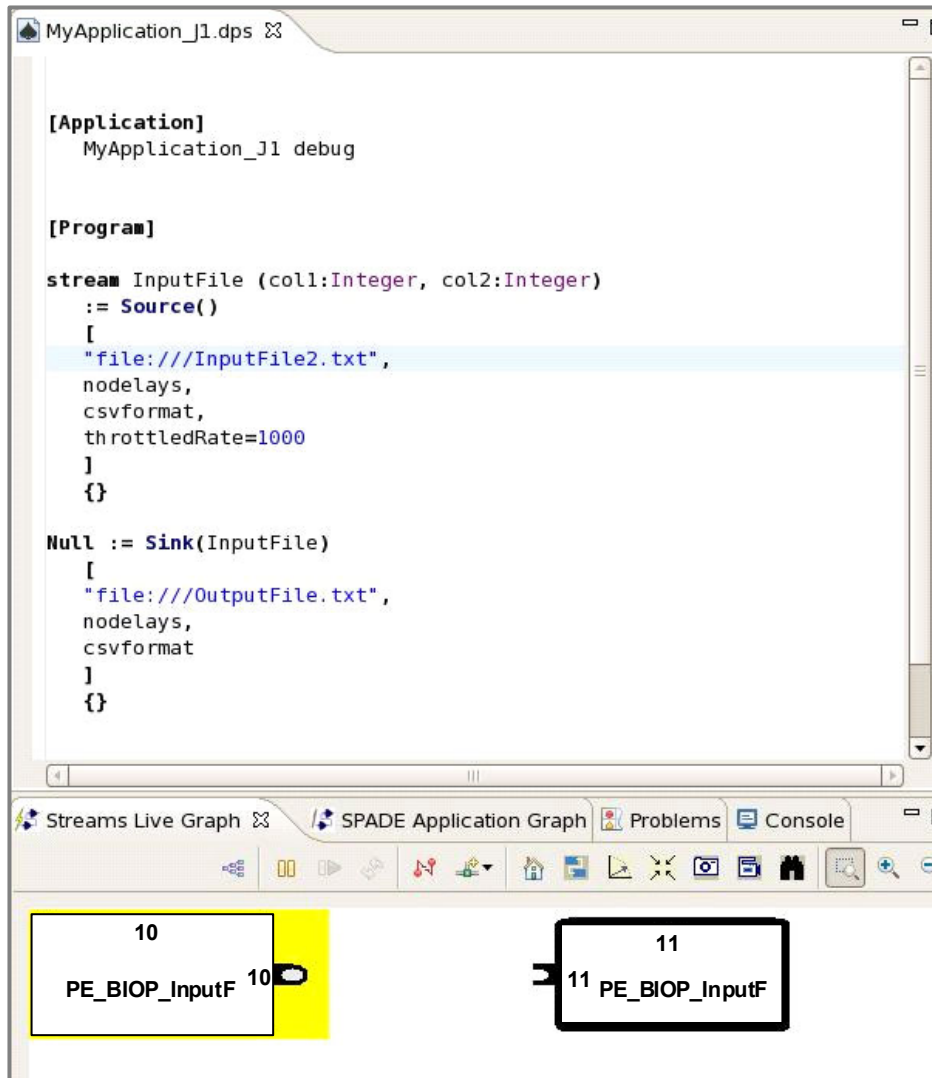


Figure 5-6 Streams Studio: Streams Live Graph, no rows being output

In Figure 5-6, we see that the MyApplication_J1 Streams Application is currently running. We know this because the Streams Live Graph View displays the Streams Application. Also in Figure 5-6, we see that there is no output from the left Operator to the one on the right. This is because the left Operator is not outputting records because the input source file was not found.

Figure 5-7 displays the Streams Live Graph View with no Streams Applications running.

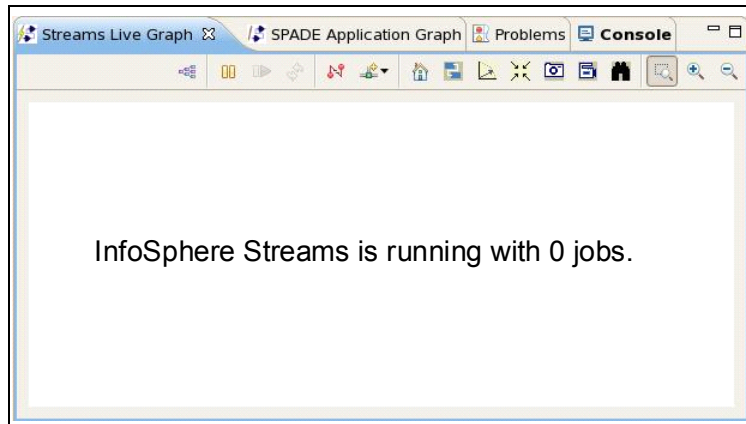


Figure 5-7 Streams Live Graph: No Application running

Figure 5-8 displays the Streams Live Graph View with the correction to the input file name. The display in the Streams Live Graph View will change color to indicate the effect of *throttledRate*, and the various pauses and then propagation of records.

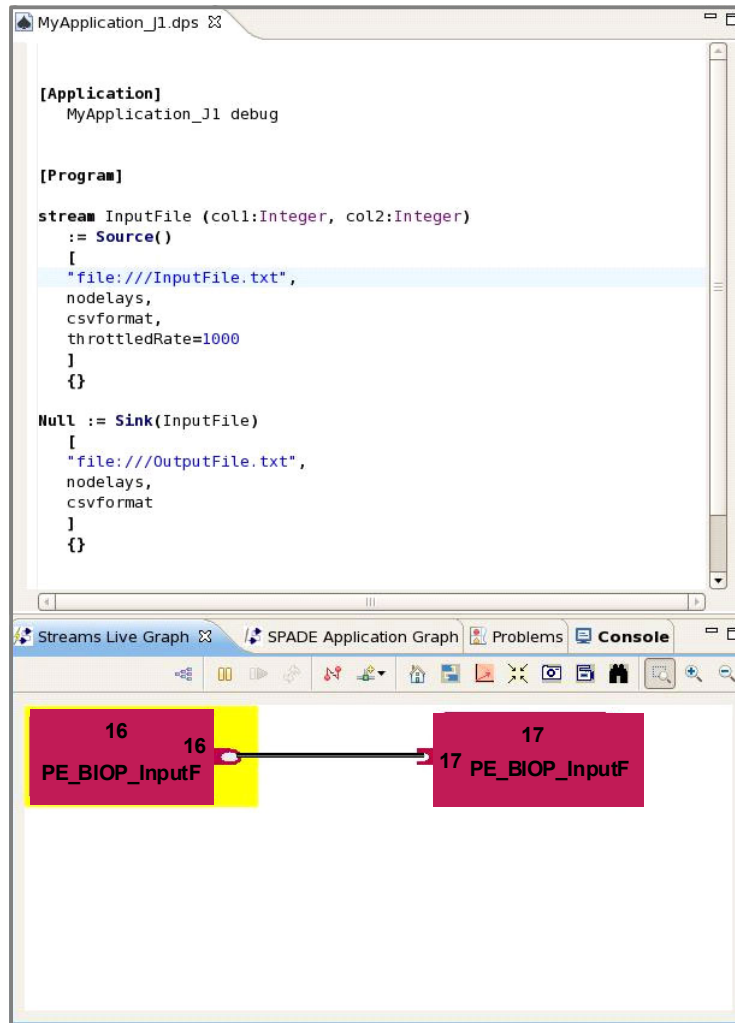


Figure 5-8 Streams Live Graph View with correction: Application running

Figure 5-9 displays the Streams Studio, Streams Live Graph Outline View, where you can observe the live progress of records consumed and sent downstream.

Identifier	Type	State	Tuples (Out/In)	Tuples In	Rate Tuples In	Tuples Out
16	PE (output only)	RUNNING	Undefined	0	0	10000
InputFile	Source	RUNNING				
17	PE (input only)	RUNNING	Undefined	10000	0	0
InputFileSink	Sink	RUNNING				

Figure 5-9 Streams Studio: Streams Live Graph View

5.6.2 Using streamtool to debug

In Example 5-40 on page 239, we added the Streams Application modifier, debug, to the Application Identifier command line. This allowed us to capture runtime diagnostics from this Streams Application in the Streams system log. Using the Streams command-line utility streamtool, we can now view this diagnostic output.

For this example, we return to the condition where the input text file name is not found, because the file does not exist. Figure 5-10 displays the first step in gathering this diagnostic output.

```

streamsadmin@redbookstreams:~
File Edit View Terminal Tabs Help
InfoSphere Streams environment variables have been set.
[streamsadmin@redbookstreams ~]$ streamtool lsinstance
spade@streamsadmin
[streamsadmin@redbookstreams ~]$

```

Figure 5-10 Using streamtool to get the Streams Instance ID

Where:

- streamtool is a command line utility, so each of these commands is entered at the operating system command line prompt.

Note: The following Streams terms are in effect here, and this section serves as a brief review:

- ▶ A *Streams Instance* refers to the collection of memory, process, and disk that is the Streams server proper. Each Streams Instance is identified by an *Instance ID*.
- ▶ The source code to a self-contained *Streams Application* exists in one operating system text file. A Streams Application is identified by its Application Name, and by its source code file name (which has the same name, with a .dps file name suffix).
- ▶ Streams Applications are submitted to run in the form of a *Streams Job*. One Streams Application may be running many copies concurrently. That is, running many concurrent Jobs of the same Streams Application.

Each Streams Job is identified by a unique Job ID.

- ▶ A Streams Job is a logical term, and exists as one or more operating system processes, referred to by Streams as *Processing Elements (PE)*. Each PE has a unique PE ID.

- ▶ In Figure 5-10 on page 245, we ran the `streamtool ls instance` command, which lists the Streams Instances, to get the list of available Streams Instance IDs.

The reported Streams Instance ID contains a special character, @. When using this value in subsequent commands, you must enclose this value inside double quotes.

In order to get the log for a given Processing Element (PE), you need the PE ID. Figure 5-11 shows the `streamtool` command that is used to get a listing of currently executing Streams Jobs and their associated Processing Elements.

```
[streamsadmin@redbookstreams ~]$ streamtool lspes -i "spade@streamsadmin"
  PeId State          RC   LC Host
JobId PeName
  18 RUNNING          -    1 redbookstreams
   9 MyApplication_J1.PE_BIOP_InputFile.1
  19 RUNNING          -    1 redbookstreams
   9 MyApplication_J1.PE_BIOP_InputFileSink1.2
[streamsadmin@redbookstreams ~]$
```

Figure 5-11 Using `streamtool` to get the PE ID

Where:

- The syntax for the streamtool command used here is:

```
streamtool lspec -i {Instance Id}
```

Where:

- lspec, entered as one word, is short for list Processing Element IDs.
- -i is followed by the Streams Instance ID shown in Figure 5-10 on page 245.
- The report in Figure 5-11 on page 246 has a two line format:
 - The Streams Job ID being is 9.
 - The Streams Job has two Processing Elements, called by its two Operators, one input and one output (see Example 5-40 on page 239).
 - The PE ID for the InputFile is 18, and the PE ID is 18.
 - If you want to view the log for the InputFile Operator, which has an error in its input file name (the file is not found), then you need a call to retrieve the log for PE ID 19. The streamtool command used to perform this call is:

```
streamtool viewlog -i "spade@streamsadmin" --pe 18
```

This command will invoke the configured operating system editor of your choice. An example is shown in Figure 5-12.

```
03 Apr 2010 02:13:53.951 [1986] DEBUG PEC M[PECServImpl.cpp:associateThread:749] P[18] - associating thread with peid=18

03 Apr 2010 02:13:53.952 [1986] DEBUG PEC M[PECServImpl.IDSIsBest.cpp:associateThread:751] P[18] - associated thread with peid=18

03 Apr 2010 02:13:53.952 [1986] DEBUG dpsop M[BIOp_InputFile.cpp:process:140]
- opening '/home/streamsadmin/workspace/MyProject_J1/data/InputFile2.txt'
each tuple has 2 attributes...

03 Apr 2010 02:13:53.953 [1986] INFO ex M[BIOp_InputFile.cpp:process:208]
- Throwing Exception: DpsOp (Msg: failed to properly open workload file
'/home/streamsadmin/workspace/MyProject_J1/data/InputFile2.txt'
(errno=2): No such file or directory)

03 Apr 2010 02:13:53.961 [1986] ERROR dpspe M[DpsSrcOpThread.h:run:48]
- Exception DPS::DpsOpException (failed to properly open workload file
'/home/streamsadmin/workspace/MyProject_J1/data/InputFile2.txt'
(errno=2): No such file or directory)
at 'virtual void DPS::BIOp_InputFile::process()'
[BIOp_InputFile.cpp:208]

Exception Code=NoMessageId with 0 substitution text strings

```

Figure 5-12 Using streamtool to get the PE log

Where:

- The fourth paragraph (denoted by white space) in Figure 5-12 starts to report the condition of the input text file not being found.
- The example in Figure 5-12 uses the operating system editor named vi. In this example, we did not make use of the syntax highlighters for this log file output.

5.6.3 Other debugging interfaces and tools

A Streams Application can have user-defined (routines) in the C/C++ programming language or Java. As Streams Studio is an Eclipse-based developers workbench, and Eclipse includes an embedded Java visual debugger, it is most common to use that area of Streams Studio to debug Java routines. There are also third-party tools that provide C/C++ language visual debuggers for the Eclipse developers workbench. Programming in C/C++ or Java, or debugging of the same, is not expanded upon further here.

A compiled Streams Application exists as an operating system binary program. Using specific Streams Application compilation arguments, the operating system

Gdb program can be used to debug Streams Applications. Debugging with *Gdb* is not expanded upon further here.

And lastly, it has been stated that a given Streams Operator can have multiple consumers of its output. For example, multiple downstream Operators can consume an upstream Operator's output. During the debugging phase of Streams Application development, it is common to insert Sink Operators merely to dump Stream contents mid-application, solely for the purpose of debugging. These command lines can be left in place, and commented out before inserting the final Streams Application into production.

Note: We particularly like to use the above technique of creating redundant (test only) Sink Operators when working with Streams Functor Operators.

With the Streams Debugger, it is extremely easy to evaluate input and output records and columns to a given Streams Operator. Seeing inside a given Functor Operator, and more specifically its flow control and variable assignments, is another challenge.

5.6.4 The Streams Debugger

Given the volume of information that has been written related to debugging a Streams Application, you might think that there was not a debugger to be found inside the Streams product, but that is not true. The Streams Debugger is currently documented in Chapter 11, “Debugging SPADE Applications”, of the *IBM InfoSphere Streams SPADE: Programming Model and Language Reference* manual. The Streams Debugger has a command-line interface, and has the following features:

- Like nearly every other debugger, the Streams Debugger offers break points, trace points, variable evaluation, and reassignment.

Generally, a break point is the ability to suspend program execution on a given condition, and a trace point is the history of a given variable, that is, how and when it changed over time.

- In Example 5-40 on page 239, we set a break point when the input record has a value of 12000. We then update this value, and continue execution of the Streams Application.

The Streams Debugger has a command-line interface, and the given Streams Application being debugged must be compiled with special arguments. As an exercise, perform the following steps:

1. Create the sample input file and sample Streams Application described in Example 5-40 on page 239.

2. From the command line, compile this Streams Application by running the following command:

```
spadec -T -g -f {Streams Application File Name}.dps
```

This command will output an operating binary file with the name {Streams Application File Name}, with no file name suffix.

3. Run the operating system binary file name listed above. This command will open the Streams Debugger command-line interface.

Note: Where are all of these files?

Given a user home directory of /home/chuck/, a Streams Project name of MyProject, and a Streams Application entitled MyApplication, the full path name to the Streams Application file name is:

```
/home/chuck/workspace/MyProject/MyApplication/MyApplication.dps
```

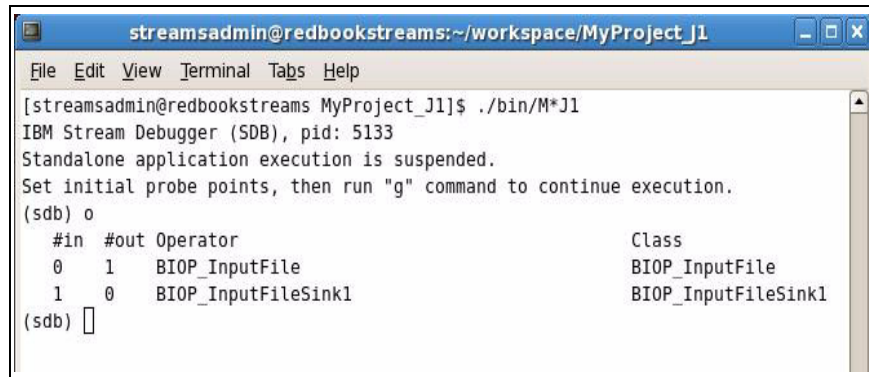
The **spadec** command will output the operating system binary program to following directory:

```
/home/chuck/workspace/MyProject/MyApplication/bin/MyApplication
```

4. Run the following Streams Debugger commands in sequence:
 - a. Call to output the Operands for this given Streams Application. Enter the following Streams Debugger command (which is a small letter “o”):

o

The output of this command is displayed in Figure 5-13.



```
streamsadmin@redbookstreams:~/workspace/MyProject_J1
File Edit View Terminal Tabs Help
[streamsadmin@redbookstreams MyProject_J1]$ ./bin/M*J1
IBM Stream Debugger (SDB), pid: 5133
Standalone application execution is suspended.
Set initial probe points, then run "g" command to continue execution.
(sdb) o
  #in  #out Operator                      Class
    0   1  BIOP_InputFile                  BIOP_InputFile
    1   0  BIOP_InputFileSink1             BIOP_InputFileSink1
(sdb) □
```

Figure 5-13 Streams Debugger: Getting the Operator ID

Where:

- The **o** (output) command lists all of the Operators for this given Streams Application. Operator Names are prefixed with a “BIOP_” label.
- The example has two Operators, an Input File Operator and a Sink Operator.

As a source (device), the Input File only lists an input, and will be referred to by its Operator Name (in this case, BIOP_InputFile), and a numeric zero.

As a destination (device), the Sink Operator only lists an output, and will be referred to by its Operator Name (in the case, BIOP_InputFileSink1), and a numeric one.

Note: Not displayed in this example is a Streams Operator with both input and output, for example, a Functor Operator.

In that case, you could refer to that Operator by its name and both a numeric zero or a numeric one. The zero would be the input side of the Operator, and one would be the output.

As certain Operators can have numerous input and output Streams, you will also see numbers here higher than one.

- In the use case being created, we want to set a break point when a given condition occurs, and our condition is when the input column on a given record is equal to the integer value of 12000.

b. Create a Break Point with the following Streams Debugger command:

```
b  BIOP_InputFile  o  0  p  $_col1  ==  12000
```

Where:

- b is the Streams Debugger command to create a new break point.
- BIOP_InputFile o 0 is the identifier of the specific Streams Operator and its specific Input Streams we want to monitor.
- p indicates we are entering a conditional break point, using the Perl expression language.

A conditional break point suspends execution of this Streams Application when its associated expression evaluates to true.

- \$_col1 is a reference to the input value of col1 from InputFile.
- == 12000 will be true when col1 equal 12000.

You must use two equal symbols here to test for equality. One equal symbol would set col1 equal to 12000, which would be true for every input record, which is not the intended goal.

Note: This evaluation expression is written in Perl, which allows many types of constructs, such as multi-level if statements and complex variable assignments.

c. Run this program by entering the following Streams Debugger command:

```
g
```

This Streams Application will run to completion, or until the conditional break point that was recorded above is encountered. An example is displayed in Figure 5-14.

```

streamsadmin@redbookstreams:~/workspace/MyProject_J1
File Edit View Terminal Tabs Help
Building PE BIOP_InputFile_BIOP_InputFileSink1.spe
Building MyApplication_J1
SPADE compilation has completed successfully, with 1 warning:
WARNING (1): You are running in operator fuse mode. Make sure: a) the operator g
raph includes no feedback links into built-in operators, and b) any feedback lin
ks into UDOPs are not connected to tuple generating inputs. Otherwise, proceed a
t your own peril!
[streamsadmin@redbookstreams MyProject_J1]$ clear

[streamsadmin@redbookstreams MyProject_J1]$ ./bin/M*J1
IBM Stream Debugger (SDB), pid: 5133
Standalone application execution is suspended.
Set initial probe points, then run "g" command to continue execution.
(sdb) o
      #in  #out Operator                                Class
      0    1  BIOP_InputFile                            BIOP_InputFile
      1    0  BIOP_InputFileSink1                      BIOP_InputFileSink1
(sdb)
(sdb)
(sdb)
(sdb) b BIOP_InputFile o 0 p $_coll == 12000
      Set + 0 Breakpoint BIOP_InputFile                                o 0
      predicate:eval($_coll == 12000) stopped:false
(sdb)
(sdb)
(sdb) g
(sdb)
      + 0 Breakpoint BIOP_InputFile                                o 0
      dropped:false predicate:eval($_coll == 12000) stopped:true
      col1, 12000, Integer
      col2, 12000, Integer
(sdb)
(sdb)
(sdb) u 0 col1 90000
      col1, 90000, Integer
      col2, 12000, Integer
(sdb)
(sdb)
(sdb)
(sdb) c
(sdb)
(sdb)
(sdb)
(sdb) q
  
```

Figure 5-14 Remainder of Streams Debugger example

Where:

- The entire example being created is shown in Figure 5-14. Extra carriage returns are entered between commands, which is allowed.
- After a **g** (go) command, the Streams Application executes until it encounters our conditional break point.

The line in Figure 5-14 on page 253 that begins with + 0 BIOP_InputFile identifies when the Streams Application suspended execution after the break point event.

At this point you could enter a variety of Streams Debugger commands, such as evaluate input columns, change the value of output columns, insert new output records, delete output records, and so on.

- The line in Figure 5-14 on page 253 that reads u 0 col1 90000 calls to update (u) the column entitled col1, in the zeroth input Stream for this Operator.

This condition is reflected in the sample output, shown in Figure 5-15.

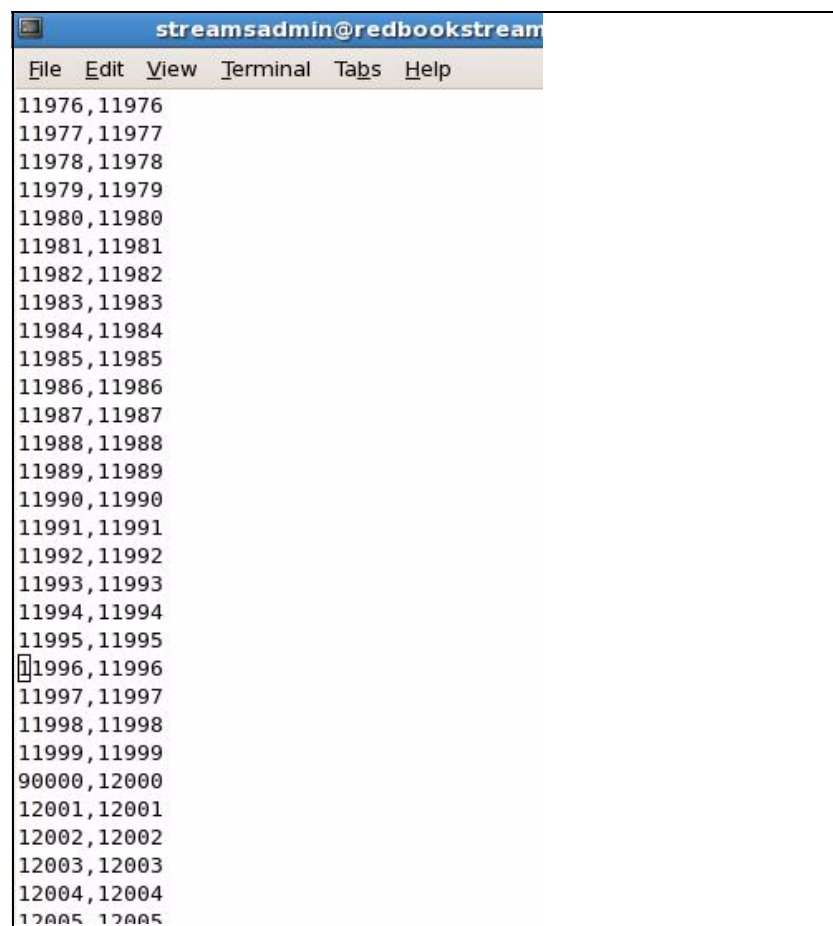


Figure 5-15 Sample output file

- In Figure 5-14 on page 253, the **c** command calls to continue execution of this Streams Application, which it does to completion.
- The **q** command exits the Streams Debugger.

Note: Why stop program execution upon receipt of the value 12000?

The point is to demonstrate that using the Streams Debugger, you can halt execution of a Streams Application on any single or set of conditions.

Why then set the new output column value to 90000?

You could use the Streams Debugger to set columns to such things as invalid values for testing or duplicates values for testing. You can also delete records from the output Streams using the **x** Streams Debugger command.

There are many, many more Streams Debugger commands than those listed here. Using the **savecfg** and **loadcfg** Streams Debugger commands, you can save and then load things such as complex sets of break points and trace points, and use them for reference or automated testing of Streams Applications.



Advanced IBM InfoSphere Streams programming

In the previous chapters, we described and discussed the concepts of streaming applications, the development process, and the fundamentals of the IBM InfoSphere Streams (Streams) Processing Language, including the 10 core Built-in Operators. In this chapter, we present advanced topics for users that are familiar with the basics of the language, can write applications with the Built-in Operators, and are ready to learn how to develop more powerful applications that take advantage the extensibility of Streams.

In 6.1, “Edge adaptors” on page 258, we cover additional Operators that facilitate the integration of Streams Applications with database servers and other network accessible data sources. These Operators are predefined and operate much like the Built-in Operators, but require more advanced configuration.

In the remaining sections in this chapter, we describe the features available to extend programs, using C++, beyond the built-in capabilities of the Streams Processing Language. With these features, you will be able to develop applications that are more readable, execute more efficiently, use existing code, and address a larger application space. Although these features are mostly independent, they are presented in increasing order of complexity and are best read in order.

6.1 Edge adaptors

Edge adaptors allow you to connect input and output streams with other systems without needing to develop any external C++ code. Using Operators from the adaptor toolkit, you can connect to IBM DB2 databases, IBM Informix Dynamic Server databases, IBM solidDB databases, market data feeds using IBM WebSphere Front Office, and text data from Internet servers (http, https, or ftp).

Operators from the adaptor toolkit follow the same general form of Built-in Operators, except that parameters are specified as a comma separated list of name/value pairs. For example:

```
Nil := ODBCAppend ( MyCompletePersonStream )
    [ connectionDocument: "connections.xml";
      connection: "PersonDB";
      access: "PersonSink" ] {}
```

With the exception of the Internet Operator, all of the Operators in the toolkit require an external connection specification to connect to the external data. Those Operators all require the three parameters shown in the above example to indicate the connection specification file and access information. The connection specification is kept in a separate file because it is typically common to many Operators, it may be complex, and it may not be the domain of the application programmer. Portions of the connection specification are described, but for complete details, see the language's reference manual.

6.1.1 DB2 and Informix Dynamic Server database Operators

The ODBCSource, ODBCAppend, and ODBCEnrich Operators provide access to DB2 and Informix Dynamic Server databases.

ODBCSource

This Operator executes an SQL SELECT statement on the external database and generates output Tuples from the resulting rows. The Attribute value of the Tuples are obtained from the columns in the resulting rows that have the same name and type. Columns with NULL values will produce a default value in the output Tuple: 0 for numeric types and "" for strings.

The actual SELECT statement is given in the connection specification file named by the connectionDocument parameter under the element named by the access parameter. This example shows how the ODBCSource is used:

```
stream LowStream ( id: Integer, item: String, quantity: Integer )
    := ODBCSource() [
```

```

connectionDocument: "configuration.xml";
connection: "InventoryDB";
access: "LowInventory";
initDelay: 3] {}

```

The optional `initDelay` specifies an initial processing delay in seconds before the Operator begins emitting Tuples. The `SELECT` statement would appear in the corresponding `configuration.xml` file as follows:

```

<access_specification name="LowInventory">
  <query query="SELECT id,item,quantity FROM inventory WHERE quantity <
  5" />
  <external_schema>
    <attribute name="id" type="Integer" />
    <attribute name="item" type="String" length="32" />
    <attribute name="quantity" type="Integer" />
  </external_schema>
</access_specification>

```

The `Attribute` elements of the `external_schema` specify the Streams data types for the columns of the `SELECT` output and must appear in the same order as they appear in the `SELECT` statement.

Additionally, `SELECT` statements in the connection specification file may be parameterized such that the `ODBCSource` Operator can customize the statement. For example, the previous example could be extended with parameters as follows:

```

stream LowStream ( id: Integer, item: String, quantity: Integer )
:= ODBCSource() [
  connectionDocument: "configuration.xml";
  connection: "InventoryDB";
  access: "LowInventory";
  threshold: 5;
  initDelay: 3] {}

```

The parameterized `SELECT` statement would appear in the `configuration.xml` file as follows:

```

<access_specification name="LowInventory">
  <query query="SELECT id,item,quantity FROM inventory WHERE quantity
  < ?" />
  <parameters>
    <parameter name="threshold" type="Integer" />
  </parameters>
...

```

The ODBC parameter marker ? is replaced by the value of the parameter named threshold provided in the ODBCSource Operator. Multiple parameter markers may appear in the SQL statement and they will be associated in lexicographical order with the parameter elements.

ODBCEnrich

This Operator enriches input streams with data from the connected database by combining input Tuples with the resulting rows and of an SQL SELECT statement performed on the database. Like the ODBCSource Operator, the SQL SELECT statement is given in the connection specification file named by the connectionDocument parameter under the element named by the access parameter. The SELECT statement can also be parameterized as described in the previous section.

Here is an example of ODBCEnrich in use:

```
stream CustomerInfo (custid: Integer, timestamp: Integer,  
    receipt: Float, daysPast: Integer, locationid :  
    Integer)  
:= ODBCEnrich ( Arriving )  
[ connectionDocument: "configuration.xml";  
  connection: "CustomersDB";  
  access:"CustomerHistory";  
  id: custid] {}
```

The corresponding access specification in configuration.xml contains the following access specification:

```
<access_specification name="CustomerHistory">  
  <query query="SELECT locationid,receipt, DAYS(CURRENT_TIMESTAMP)-  
DAYS(date) as daysPast FROM cust_sales WHERE ID = ?" AND daysPast  
< 90 />  
  <parameters>  
    <parameter name="pid" type="Integer" />  
  </parameters>  
...
```

In this example, the arriving stream contains the customer ID (custid) and time stamp of arriving customers. The Operator invokes the SELECT statement to retrieve the recent purchase history of the customer. The SELECT is executed for every incoming Tuple (note that the custid Attribute of the incoming Tuple is used as a parameter to the SELECT statement). The SELECT results may contain multiple rows and the output Tuples are produced as a Cartesian product of the incoming Tuple with the resulting row set.

The Attributes of the output stream must match the name and type of either an input stream Attribute or a column of the resulting row set, unless they are assigned in an output Attribute assignment.

ODBCAppend

This Operator stores a stream of Tuples in the associated database, inserting one row per Tuple. For every column in the target database table, there must be an Attribute of the same name and type in the input stream of the ODBCAppend Operator.

The rows are inserted into the database as specified in the connection specification file named by the connectionDocument parameter under the element named by the access parameter. For example, the Operator is used as follows:

```
Nil := ODBCAppend ( AlertMessages)
      [ connectionDocument: "configuration.xml";
        connection: "AdminDB";
        access: "AlertInsertion" ] {}
```

The corresponding configuration.xml file would contain an access_specification element, such as:

```
<access_specification name="AlertInsertion">
  <table tablename="alerts" transaction_batchsize="10" rowset_size="4"
  />
...
```

The table element names the table that the data is to be inserted into and two optional Attributes. The rowset_size Attribute specifies the number of rows sent to the database server per network flow, and the transaction_batchsize Attribute specifies the number of rows to commit per transaction, which must be greater than or equal to the rowset_size.

6.1.2 The solidDBEnrich Operator

The solidDBEnrich Operator performs the same function as the ODBCEnrich Operator, except that it executes a solidDB table query rather than the more general SQL SELECT statement. The solidDB table query is more restricted, but is optimized for high performance. The query performance may be significant to overall application performance because the query must be performed for every incoming Tuple.

The table query is specified in the `access_specification` of the associated `connectionDocument`. For example:

```
<access_specification name="AnalystRating">
  <tablequery tablename="ratings">
    <parameter_condition column="symbol" condition="equal" />
    <parameter_condition column="type" condition="equal" />
  </tablequery>
  <parameters>
    <parameter name="ticker" type="String" />
    <parameter name="ttype" type="String" />
  </parameters>
</access_specification>
```

This example does a query against the ratings table for rows whose symbol and type column are equal to the ticker and type parameters, respectively. The results of the query are combined with incoming Tuples to form output Tuples in the same fashion as the ODBCEnrich Operator.

6.1.3 The WFOSource Operator

The WFOSource Operator generates a stream from market data feeds available from WebSphere Front Office (WFO). Again, the Operator's behavior is governed by the `access_specification` element of the external `connectionDocument`. For WFO messages that match an object subscription and whose event type is a market data event or event summary, the Operator automatically assigns the values of the fields in the message to the output stream Attributes with the same name and data type.

An example connection specification looks like the following:

```
<access_specification name="StockTicker">
  <feed service_name="NUTP" object_name="AAPL,MSFT"
    output_attribute_for_object_name="item"
    output_attribute_for_event_type="type">
    <event_type name="CONTRIBUTE" />
    <event_type name="TRADE" />
    <event_type name="QUOTE" />
  </feed>
  ...
</access_specification>
```

The feed element specifies the name of the WFO service to connect to (`service_name`), the object streams to subscribe to (`object_name`), and optionally, Attributes to receive certain metadata (`output_attribute_for_object_name` and `output_attribute_for_event_type`). The `event_type` elements specify the names of the types of events that will be selected from the object streams for output.

6.1.4 The InetSource Operator

The InetSource Operator generates output Tuples from one or more text-based data feeds accessible via the http, https, ftp, or file protocols. The basic operation is to fetch data from one or more URIs at a given interval and output either the entire content or only the new content added since the last fetch. The InetSource Operator does not attempt to parse the data in any way, and the output stream schema may define only one Attribute of type String or StringList. Parsing can be done by a downstream Operator, such as a Functor.

The following is an example InetSource Operator:

```
stream WxObservations ( metarObsRecord: String )
:= InetSource (
  [ URIList:
    "http:// noaa.gov/pub/data/observations/cycles/07Z.TXT",
    "http:// noaa.gov/pub/data/observations/cycles/08Z.TXT";
    initDelay: 5;
    incrementalFetch;
    fetchIntervalSeconds: 60
  ] {}
```

The InetSource Operator accepts several configuration parameters:

- ▶ **URIList:** A comma separated list of URIs in quotes with a protocol specification of http, https, ftp, or file. All URIs are fetched at the given interval and each one may produce output.
- ▶ **initDelay:** An initial delay in seconds before the Operator begins fetching data for the first time and generating output.
- ▶ **fetchIntervalSeconds:** The interval at which the data sources will be fetched and checked for output. However, the URI is first checked for a last modified time, and if that has not changed, then the data is not fetched and there is no output from that URI.
- ▶ **unconditionalFetch:** When present, data is fetched and processed according to fetchIntervalSeconds, even if the last modified time of the resource has not changed.
- ▶ **inputLinesPerRecord:** A number specifying how many rows of the fetched text make up a record. If inputLinesPerRecord is not specified, each line will be a unique record. If inputLinesPerRecord is more than one, then the specified number of lines are combined together with a separator (see intraRecordPadValue) to form a single record.
- ▶ **intraRecordPadValue:** Specifies the separator to use when combining multiple lines into a single record. This Attribute must be double quoted. If intraRecordPadValue is not present, the space character is used.

- ▶ `incrementalFetch`: When present, this Attribute indicates that only new records added since the last fetch are output. The default behavior, when `incrementalFetch` is not present, is to split the fetched data into records and output each record as a Tuple.
- ▶ `doNotStreamInitialFetch`: When present, no output will be generated from the initial data fetch. This Attribute, used together with `incrementalFetch`, prevents the generation of output for all the existing data at the time of application startup. Instead, only new data added since the application began will be output.

6.2 User-defined functions

User-defined functions are externally defined functions written in C++ that can be invoked from within the Streams Processing Language. These functions can serve many purposes such as:

- ▶ Encapsulating related calculations
- ▶ Reusing common computations
- ▶ Invoking external libraries
- ▶ Improving performance
- ▶ Providing functionality not available in Streams

They are simple to incorporate into a Streams Application, and can be used anywhere expressions appear, such as output Attribute assignments, or custom logic of the Functor Operator.

You typically define the function body in a header file that will be included in the Streams Application via the package statement. For example,

```
[Application]
fir
```

```
[Libdefs]
package "FirFunctions.h"
```

The package statement appears in the Libdefs section of the Streams file, which appears right after the Application section. You are free to choose any name for the file, but the compiler will look in the `optsub` directory of the project for it. Of course, you can have multiple package statements to include multiple files. If you have many functions, you can organize them into separate files.

The header file should include the complete implementation of your function(s) and each function must be preceded with a specially formatted comment that serves as a prototype to the Streams Application.

For example, the `FirFunctions.h` file contains:

```
#ifndef FIR_FUNCTIONS_H
#define FIR_FUNCTIONS_H

/// @spadeudf Short Dsp::Fir12(ShortList)

namespace Dsp {
    static int16_t coef[12]={4,0,1,1,1,1,1,1,1,1,0,4};

    int16_t Fir12(dlist<int16_t> window)

    {
        int32_t accum = 0;
        for (uint32_t tap = 0; tap < 12; tap++) {
            accum += window[tap] * coef[tap];
        }
        return (int16_t)(accum>>2);
    }
}

#endif
```

As seen above, the special comment begins with `/// @spadeudf` and is followed by the Streams function prototype. The prototype specifies the return type, function name, and parameter types using the Streams Processing Language type names. This example shows how the function can be defined within a C++ name space, which is recommended to avoid possible name conflicts with the many other parts of the application.

Notice in the `Fir12` function definition above that there are predefined C++ types that correspond to the built-in types provided by the Streams Processing Language. In Table 6-1, we summarize the Stream Processing Language types and their corresponding C++ types. List types are implemented in C++ with the `dlist` template class. For example, the `StringList` type corresponds to the `dlist<std::string>` type in C++.

Table 6-1 Streams Processing Language types and corresponding C++ types

Streams Processing Language	C++
Byte	int8_t
Short	int16_t
Integer	int32_t
Long	int64_t

Streams Processing Language	C++
Float	float
Double	double
String	std::string
Boolean	bool

The Fir12 function implements a finite-response filter that is commonly used in DSP applications to filter a stream of data. It is a Sliding window algorithm producing one new value for every input value computed from a window of past values. The following example shows how the Fir12 function is called from the Streams Application:

```
stream Fir(Filtered : Short)
:= Functor(Waveform)
<
  ShortList $history := [0s,0s,0s,0s,0s,0s,0s,0s,0s,0s,0s,0s];
>
<
  $history := remove($history, 0);
  $history := insert($history, Value, size($history));
>
[true]{Fir12($history)}
```

The input stream Waveform contains a single Attribute Value of type Short. The Functor keeps a history of the past 12 values received in a ShortList, which is passed into the Fir12 function to produce an output value.

6.2.1 Using templates

User-defined functions can also be implemented with a C++ template to make them type generic. To use a template function, simply precede the template with a Streams prototype for each form you plan to use in the Streams Application. For example, the Fir12 function could have been implemented with a template as follows.

```
/// @spadeudf Float Dsp::Fir12(FloatList)
/// @spadeudf Long Dsp::Fir12(LongList)

namespace Dsp {
  static int16_t coef[12]={4,0,1,1,1,1,1,1,1,1,0,4};

  template<class _T> _T Fir12(dlist<_T> window)
```

```

{
    _T accum = 0;
    for (uint32_t tap = 0; tap < 12; tap++) {
        accum += window[tap] * (_T)coef[tap];
    }
    return(accum/4);
}
}

```

This version can be used with either the Float or Long type within the Streams Application.

6.2.2 Using libraries

If your function makes use of a nonstandard library or the function itself is defined in a library, then you must specify the library name and path in the Streams Application. The Libdefs section may contain one or more libpath, and libs statements as follows:

```

[Libdefs]
libpath "/home/streamuser/lib"
libs "gzlib" "opra"

```

The libpath statement adds a directory to the library search path, and the libs statement adds one or more libraries to be linked with the application. Notice that the lib prefix and extension of the library are omitted (for example, gzlib rather than libgzlib.so).

The user-defined function may itself be defined in a library, but you must still provide the specially formatted comment prototype in a header file included via the package statement.

6.3 User-defined Source Operators

Source Operators allow a Streams Application to input data from external systems or storage. The Built-in Source Operator described in Chapter 5, “IBM InfoSphere Streams Processing Language” on page 187 can handle data in text or binary format, and can access data from files and TCP/UDP sockets in a number of ways. In many cases however, existing data formats will not fit the requirements of the Built-in Operator.

It is possible to create an external application that parses and sends the data to the Streams Application in the accepted format via a socket, but that complicates the application and could have a performance impact.

The user-defined Source Operator enables you to write custom C++ code to read and parse input data to create one or more output streams. Although you could achieve the same goal with the general user-defined Operator, using a user-defined Source Operator allows you to take advantage of the specialized features of the Source Operator, such as protocol handling and compression.

6.3.1 Example user-defined Source Operator

Let us now look at a simple text file example. Web servers record information about every access to a website in a text format like the following:

```
88.211.17.146 - - [27/May/2010:04:03:44 -0700] "GET /crossdomain.xml
HTTP/1.1" 200 207 "-" "Mozilla/5.0 (compatible; U; Chumby; Linux)
Flash Lite 3.1.5"
88.211.17.146 - - [27/May/2010:04:03:44 -0700] "GET /fcgi-
bin/chumby.fcgi?action=votd&un=davy3c HTTP/1.1" 200 335 "-"
"Mozilla/5.0 (compatible; U; Chumby; Linux) Flash Lite 3.1.5"
```

This log data includes the originating IP address, the time of the request, the requested resource, the result code, result size, referrer, and user agent. Although the format is simple enough, it is not compatible with the standard .csv format required by the Built-in Source Operator.

We are going to create an example that monitors the server load in terms of the number of users and bandwidth. A user-defined Source Operator is defined by simply added the `udfTxtFormat` or `udfBinFormat` option, as shown in this example:

```
stream Access(
    ip: String,
    time: Integer,
    size: Integer)
:= Source()
["file:///access_log",udfTxtFormat="LogParser"] {}
```

The `udfTxtFormat` option specifies the name of the custom module that will implement the Operator (`LogParser` in this example).

When you invoke **make** for the first time to build an application containing user-defined Operators, skeleton source code is generated that defines a single C++ class to implement that Operator. You then modify the generated files to add your custom parsing of the input data. These files are only created if they do

not already exist, so your changes will never be overwritten. Of course, you can remove the files and run **make** again to start over.

The skeleton code is generated in the `src` subdirectory of the project directory, using the module name as a prefix. For this example, the generated files are:

1. `LogParser_includes.h`: Contains extra `#include` statements required by your custom code.
2. `LogParser_members.h`: Contains the member declarations of the `Operator` class.
3. `LogParser.cpp`: Contains the implementation of your member functions.

Together, these files define the C++ class that is always named `SourceOperator` (even if you have multiple user-defined Source Operators). The `SourceOperator` class must include at least the following three member functions (where `<module>` is the name of your module (`LogParser` in this example)):

1. `void <module>_initialize(void)`: This member is invoked only once when the application starts, and can be used to allocate or initialize member variables. For example, you may need to load configuration data, initialize counters, and so on.
2. `void <module>_finalize(void)`: This function is invoked only once when the application shuts down and can be used to deallocate data or do other cleanup.
3. `void <module>(char const* buffer, const unsigned buflen)`: This is the main function of the custom `Operator`. This function is invoked with each line from the input data source as it is received or read. Although the function is invoked line-by-line, you are not required to output `Tuples` every time it is called, so there is no restriction about how data is organized into lines.

The skeleton files include definitions of the three required member functions with empty bodies in the `<module>_members.h` file. There are also three definitions of the same functions in `<module>.cpp` with comments around them. The generated comments suggest that if any of the functions are simple (for example, one-liners), you can just insert the code into the bodies in the `<module>_members.h` file. However, you will normally want to place your custom code in the `<module>.cpp` file, as we do in this example, which means you need to delete the empty body in the `<module>_members.h` file, make it into a prototype, and then un-comment the function in the `<module>.cpp` file.

The web server example does not require any state to be maintained between processing each line, so the `LogParser_initialize` and `LogParser_finalize` functions are empty. The following shows the complete implementation of the main `LogParser` function, and is explained below:

```
void SourceOperator::LogParser(char const* buffer, const unsigned
buflen)

{
    char ip[16];
    int fields[8];
    struct tm time;

    OPort0_t otuple;

    // Parse line into fields, field positions output in array 'fields'
    splitLogLine(buffer,buflen,fields,7);

    // IP address
    parseIp(buffer+fields[0],buflen,ip);
    otuple.set_ip(std::string(ip,8));

    // Time
    memset(&time,0,sizeof(time));
    strptime(buffer+fields[3]+1,"%d/%b/%Y:%H:%M:%S",&time);
    otuple.set_time(mktime(&time));

    // Size
    otuple.set_size(atoi(buffer+fields[6]));

    submitTuple0(otuple);
}
```

The parameters of the main function are always the same and consist of a pointer to a line of text and the length of the line. For the purpose of simplification, the `LogParser` input is assumed to be well formed. The function body parses the text and generates an output Tuple from the data.

The LogParser method declares a variable of the OPort0_t type to contain the output Tuple. The compiler automatically creates a C++ class named OPort0_t based on the Operator's output stream schema. The class includes accessor methods that are used to fill the Tuple with output values. In this case, the output stream type is (ip: String, time: Integer, size: Integer), so the OPort0_t class includes the methods:

```
▶ void set_ip(std::string);  
▶ void set_time(int32_t);  
▶ void set_size(int32_t);
```

The parameter types are the C++ equivalents of the Streams Processing Language types (see Table 6-1 on page 265).

After filling the Tuple variable with data, it must be submitted to the output stream for it to be passed downstream in the application. The Tuple is passed to the built-in function submitTuple0 to submit it.

The example above makes use of a couple of auxiliary functions that do the parsing. The splitLogLine function parses the line into fields, storing the location of each field in the local fields array. The parselp function parses an IP address in dot notation (192.168.1.1) and produces a hexadecimal string representation. These functions are not large, so they are included in the LogParser.cpp file, and defined as members of the class in LogParser_members.h. If you require a lot of code, or define other C++ classes, then you may want to create a library instead. The user-defined Functions section describes how to specify libraries that should be linked with the application.

6.3.2 Multiple streams

It may be the case that your data source contains data for multiple streams of your application. You can pass all the data into the Streams Application and use Built-in Operators to split it into multiple streams, but it may be awkward if the streams do not share a lot of common fields. It is simple to define a user-defined Source Operator with multiple output streams, as shown in this example:

```
stream Access(  
    ip: String,  
    time: Integer,  
    size: Integer)  
stream Referral(  
    time: Integer,  
    source: String,  
    request: String)  
stream Invalid(  
    ip: String,
```

```

time: Integer,
request : String,
client : String)
:= Source()
["file:///access_log",udfTxtFormat="LogParser"] {}

```

Multiple output streams are defined by including multiple stream definitions preceding a single `:=`. When there are multiple output streams, the compiler will generate multiple Tuple classes to represent them numbered from 0 on. In this example, the three Tuple classes, `OPort0_t`, `OPort1_t`, and `OPort2_t`, are generated to represent the streams in the order that they appear in the source code (Access, Referral, and Invalid, respectively).

Likewise, three submit functions are available to submit Tuples to each of the three streams: `submitTuple0`, `submitTuple1`, and `submitTuple2`. You are not required to output Tuples for each stream every time the line processing function is called. For example, if the input stream contains records of various types, then you can define one output stream for each record type and output each record to the corresponding output stream.

6.3.3 Unstructured data

Streams also supports the ability to attach a payload to Tuples that is not manipulated by Built-in Operators, but propagates through the application and can be accessed by user-defined Operators, and user-defined Sink Operators. For example, there may be image or other media associated with each Tuple that is not processed by the application, but is attached to the Tuple so that it may be stored together with the Tuple in the application's output. See 6.5, “User-defined Operators” on page 276 for details about using payloads.

6.4 User-defined Sink Operators

The Built-in Sink Operator allows you to output streams to external storage or applications in standard .csv format or binary record formats (see Chapter 5, “IBM InfoSphere Streams Processing Language” on page 187). User-defined Sink Operators enable you store or transmit data in a custom format using your own formatting code written in C++. Streams will still handle all the I/O involved for the chosen protocol (file, UDP, and so on), so you only need to implement the data formatting.

6.4.1 Example user-defined Sink Operator

The example presented in this section is a Sink Operator that accepts incoming Tuples that represent scan lines of an image, and writes that image to a file in a simple run-length encoded format (PCX). A user-defined Sink Operator is defined by adding the `udfBinFormat` or `udfTxtFormat` option of the Sink Operator. For this example, the Operator is defined as follows:

```
Nil := Sink(Scans)
      ["file:///globe.pcx",udfBinFormat="pcxFormatter"] {}
```

The Scans stream has the type (scan: ByteList). Each Tuple of the input stream contains one scan line of 8-bit pixel data.

The `udfBinFormat` option defines the name of the formatting module, which is `pcxFormatter` in this example. When you compile the application with the user-defined Sink Operator for the first time, the compiler will generate skeleton source code files for the Operator. You can then edit the files to implement the custom formatting. The compiler will never overwrite these files if they already exist, but you can delete them and re-run the compiler to start over.

For this example, the compiler will generate these files using the module name as a prefix:

- ▶ `pcxFormatter_includes.h`: Contains extra `#include` statements required by your custom code.
- ▶ `pcxFormatter_members.h`: Contains the member declarations of the custom class.
- ▶ `pcxFormatter.cpp`: Contains the implementation of your member functions.

When the module is instantiated by the compiler, many standard headers are included, so you will not typically need to add anything to the `_includes.h` file, unless you need to make use of a non-system library.

Together, the three generated files define a single C++ class for the user-defined Sink Operator. The C++ class requires at least the following three methods, prefixed by the module name (`pcxFormatter` in this example) that must be declared in the `<module>_members.h` file and implemented in the `<module>.cpp` file:

- ▶ `void <module>_initialize(void)`: Invoked once during application startup, this method should initialize and allocate any member variables.
- ▶ `void <module>_finalize(void)`: Invoked once after all Tuples have been received, this method can be used to clean up or free data used by the class.

- `void <module>(const IPort0_t& ituple, char*& buffer, unsigned& buflen)`: This is the main formatting method that will be invoked for every incoming Tuple to format the output.

For the following example, `pcxFormatter_methods.h` contains two member variable declarations in addition to the required methods:

```
int scanline;
unsigned char outbuf[1024];

void pcxFormatter_initialize(void);

void pcxFormatter_finalize(void);

void pcxFormatter(const IPort0_t& ituple, char*& buffer, unsigned&
buflen);
```

The `scanline` variable will keep track of what line of the image is being formatted. The `outbuf` variable is used as a temporary buffer to hold output data as described below.

As for all user-defined Sink Operators, the `pcxFormatter` method has one input parameter, `ituple`, and two output parameters, `buffer` and `buflen`. This method is called once for every Tuple received by the Sink Operator with that Tuple as the first argument. The `ituple` parameter is of the `IPort0_t` type, which is an automatically generated class that represents Tuples of the input stream, including accessor methods to retrieve the values of each Tuple Attribute. In the `pcx` example, the input stream has the type (`scan: ByteList`), and the `IPort0_t` class has the method `const dlist<int8_t> get_scan()`.

The `pcxFormatter` method is responsible for managing its own memory for output data. Before returning, the method must assign the output parameter `buffer` with a pointer to the output data, and `buflen` with the length of data to be output. Note that this data will be used after the `pcxFormatter` method returns, so you cannot use a local variable for the output data. You must use a class member variable or dynamically allocated memory. If dynamically allocated memory is used, then you are responsible for de-allocating it.

The `pcxFormatter` is implemented as follows:

```
void SinkOperator::pcxFormatter(const IPort0_t& ituple, char*& buffer,
unsigned& buflen) {
    char last;
    unsigned i,n,cnt;
    unsigned pos=0;
    const dlist<int8_t> data=ituple.get_scan(); // get ByteList from
ituple
```

```

buffer=(char *)outbuf; // set output parameter buffer to our buffer

if (scanline==0) // write header first time through
    pos=fillHeader(outbuf,data.size(),NUM_SCANLINES);

// generate run-length encoded data for the ByteList
n=data.size();
cnt=1;
last=data[0];
for (i=1; i<n; i++) {
    if (data[i]==last)
        cnt++;
    else {
        outbuf[pos++]=0xc0|cnt;
        outbuf[pos++]=last;
        last=data[i];
        cnt=1;
    }
}
outbuf[pos++]=0xc0|cnt;
outbuf[pos++]=last;

scanline++;
if (scanline==NUM_SCANLINES) // append palette after last scanline
    pos+=fillPalette(outbuf+pos);

buflen=pos; // set output parameter buflen to number of bytes
}

```

The PCX format output by this example requires both header and footer data. However, there is no mechanism to output data in either the `<module>_initialize` or `<module>_finalize` methods. All data must be output by the single `pcxFormatter` method. Our implementation uses the `scanline` variable to output a header when the first scan line is received and output the footer when the last scan line is received. Because the method can return only one block of data for output, the header and footer must be combined with the current scan line data for output.

The `outbuf` member variable is used to hold the output data, which is less complicated than using dynamically allocated memory. However, you must be careful to avoid buffer overruns. A safer approach might be to use a `std::vector` variable.

6.4.2 Unstructured data

Tuples may also have unstructured data attached to them as payloads. The user-defined Operators section describes how to attach a payload to a Tuple. User-defined Sinks can access the payload attached to a Tuple for output. See 6.5, “User-defined Operators” on page 276 for details about accessing payloads.

6.5 User-defined Operators

The user-defined Operator is a custom Operator similar to the user-defined Source and Sink Operators, except it has both input and output streams. Also, unlike the user-defined source and Sink, the user-defined Operator has its own Operator name, that is, UDOP. In this section, we implement a more powerful version of the Fir filter presented in 6.4, “User-defined Sink Operators” on page 272 by using a user-defined Operator.

Note: In this section, we only address C++ UDOPs. For details about Java UDOPs, refer to the *Streams Language Reference* manual.

6.5.1 An example user-defined Operator

The UDOP Operator looks similar to other Operators, but the semantics are slightly different. The following example demonstrates the UDOP syntax:

```
stream Fir(Filtered : Integer)
:= Udup(Waveform)
["FirFilter"]
{coefficients="4,0,1,1,1,1,1,1,1,0,4"}
```

The output streams appear before the :=symbol and the input streams appear within parenthesis like the other Built-in Operators. However, unlike other Operators, the square brackets contain the name of the C++ module that implements the user-defined Operator. The module name is used to generate file names and the class name. The curly braces contain arbitrary options that are passed to the C++ module and can be used for any purpose. For this example, the coefficients option provides the coefficient values for the filter and determines the size of the filter window.

Implementing user-defined Operators

The overall data flow of a custom Operator is shown in Figure 6-1. The Operator is implemented by a C++ class, whose processInputN methods process input Tuples and submit output Tuples via the submit0 method. You are responsible for implementing the processInputN methods that define the behavior of the Operator. The submit0 method is automatically generated.

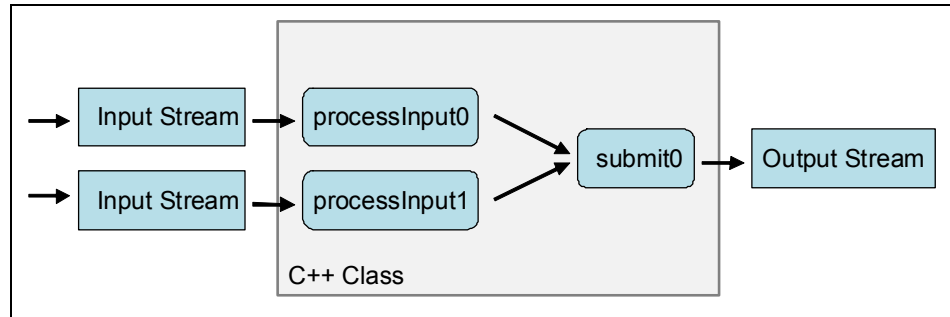


Figure 6-1 Overall data flow of a custom Operator

As with the user-defined Source and Sink Operators, when you compile an application for the first time with a UDOP, the compiler will generate skeleton files in the project's src subdirectory. There are three files that you will edit to implement the custom Operator, and the compiler will not overwrite them if they already exist. The file names are prefixed with "UDOP_" and the module name.

For the example above, the compiler generates the following files:

- ▶ UDOP_FirFilter_includes.h: Contains extra #include statements required by your custom code.
- ▶ UDOP_FirFilter_members.h: Contains the member declarations of the custom class.
- ▶ UDOP_FirFilter.cpp: Contains the implementation of your member functions.

Together, these files define and implement the UDOP_FirFilter class. There will also be a UDOP_FirFilter.h file in the src subdirectory, but it should not be edited and will be overwritten every time the application is compiled.

The generated files contain skeleton code to help you get started implementing the Operator. The cpp file contains several standard member functions with empty function bodies to be implemented. The standard members are:

- ▶ void initializeOperator(): This member is invoked only once when the class is instantiated and can be used to allocate or initialize member variables.

- ▶ `void processCmdArgs(const string& args)`: This member is invoked only once before the Operator begins accepting Tuples and can be used to process the Operators custom options and perform initialization.
- ▶ `void finalizeOperator()`: This function is invoked only once when the application shuts down and can be used for cleanup.
- ▶ `void processInput0(const IPort0_t & tuple)`: This is the main function of the custom Operator. It is invoked once for every input Tuple received and can generate zero, one, or more output Tuples. If there is more than one input stream, then there will be a `processInputN` method for each input stream.

Unlike the user-defined Source and Sink Operator, these methods are automatically defined and do not need to be included in the `UDOP_<module>_members.h` file.

Custom member variables and functions

You can add declarations for any custom member variables or functions in the generated `UDOP_<module>_members.h` file. For the FIR example, the `UDOP_FirFilter_members.h` file contains:

```
int32_t nTaps;
int32_t *coef;
int32_t *window;

void setCoefficients(const char *coefstr);
```

The `nTaps` member is the size of the Sliding window, `coef` is an array of coefficients, and `window` is an array of the most recently received values.

The `setCoefficients` function is specific to this example, and is used to parse the coefficients option.

Initialization and finalization

The initialization method in the FIR example merely sets the member variables to some default values. Further initialization is performed in the `processCmdArgs` method.

The finalization method simply releases dynamically allocated memory.

These methods are defined in `UDOP_FirFilter.cpp` as follows:

```
void UDOP_FirFilter::initializeOperator()
{
    nTaps=0;
    window=coef=NULL;
}
```



```
void UDOP_FirFilter::finalizeOperator()
{
    if (window!=NULL) delete window;
    if (coef!=NULL) delete coef;
}
```

Operator configuration

The processCmdArgs method is called to process the UDOP's options within curly braces. The options are rewritten by the compiler into a command-line style string where the option name is preceded by "--", followed by a space and then the option value. From the example above, the {coefficients="4,0,1,1,1,1,1,1,1,0,4"} options are transformed into the string "--coefficients 4,0,1,1,1,1,1,1,1,0,4", which is passed as an argument to processCmdArgs.

In the UDOP_FirFilter.cpp file, we implement processCmdArgs as follows:

```
void UDOP_FirFilter::processCmdArgs(const string& args)
{
    int i;
    ArgContainer arg(args);

    for (i=0; i<arg.argc; i++)
        if (strcmp(arg.argv[i], "--coefficients")==0) break;
    if ((arg.argc-i)<2)
        THROW(DpsOp, "Missing coefficients command line switch.");
    setCoefficients(arg.argv[i+1]);
}
```

This implementation uses the helper class ArgContainer to parse the args string into an array of strings. To use ArgContainer, you must include UTILS/ArgContainer.h in the cpp file. The loop searches for the coefficients option and then calls setCoefficients with the corresponding value.

The custom setCoefficients method parses the coefficient values, filling the coef array. It also sets the value of nTaps and initializes the window array to zeros.

Processing Tuples

The main functionality of the Operator is to process Tuples, and that is done in the processInput0 method. This method is called each time a Tuple is received from the input stream, and in this example produces one output Tuple.

The processInput0 method of the FIR example is:

```
void UDOP_FirFilter::processInput0(const IPort0_t & tuple)
{
```

```

int tap;
int32_t accum=0;

for (tap=1; tap<nTaps; tap++)
    window[tap-1] = window[tap];
window[nTaps-1] = tuple.get_Value();

for (tap=0; tap<12; tap++) {
    accum += window[tap] * coef[tap];
}

OPort0_t otuple;
otuple.set_Filtered(accum>>2);
submit0(otuple);
}

```

The input Tuple is passed as the Tuple argument of the method, and the method submits output Tuples by invoking the predefined submit0 method.

The input Tuple has the type IPort0_t, an automatically generated C++ class that represents Tuples of the input stream Waveform. The IPort0_t class will have one accessor function for each Attribute of the input stream with the same name as the Attribute plus the "get_" prefix. The Waveform Tuples have a single Attribute Value of type Integer, so the IPort0_t in this example include a uint32_t get_Value() method. The C++ types associated with Streams Processing Language types are shown in Figure 6-1 on page 277.

Similarly, the OPort0_t class have one setter method for each Attribute of the output stream with the same name of the Attribute plus the "set_" prefix. Note that it is not enough to declare the output Tuple and set all its Attributes; you must invoke the submit0 method to output the Tuple on the output stream.

6.5.2 Multiple streams

As with Built-in Operators, user-defined Operators can also have multiple input and output ports. There will be one processInputN method for each defined input port, which will take as an argument a Tuple of type IPortN_t, where N is 0, 1, 2, and so on.

Likewise, for each defined output port, there will be one submitN method, which will take as an argument a Tuple of type OPortN_t, where N is 0, 1, 2, and so on.

For example, consider the following user-defined Operator:

```
stream ChannelOne(x : Integer)
stream ChannelTwo(y : Integer)
stream ChannelThree(z : Integer)
:= Udop(LeftIn,RightIn)
  ["CrossOp"]
  {}
```

The generated CrossOp class in this example would have two input process methods:

- ▶ void UDOP_CrossOp::processInput0(IPort0_t ituple): Invoked with LeftIn tuples
- ▶ void UDOP_CrossOp::processInput1(IPort1_t ituple): Invoked with RightIn tuples

There would also be three submit methods:

- ▶ void UDOP_CrossOp::submit0(OPort0_t otuple): Submit Tuple to ChannelOne stream
- ▶ void UDOP_CrossOp::submit1(OPort1_t otuple): Submit Tuple to ChannelTwo stream
- ▶ void UDOP_CrossOp::submit2(OPort2_t otuple): Submit Tuple to ChannelThree stream

6.5.3 Unstructured data

Every Tuple in a Streams Application is capable of carrying an unstructured payload that is generally passed along by the Built-in Operators, but can be used for any purpose by user-defined Operators. The payload is simply a block of dynamically allocated memory containing arbitrary data (text, audio, custom format, and so on) that is associated with a Tuple.

Note that you can also pass arbitrary data with a Tuple as a ByteList Attribute, which is the recommended method. Because payloads are not explicit in the Streams program, they are easy to neglect when modifying the program, which could lead to errors. However, if you have a large chunk of data, then there might be a performance issue creating a ByteList object, and you may choose to use a payload.

The Built-in Operators treat payloads as follows:

- ▶ Sort, Functor, Puncctor, Delay, Split, Source, and Sink: These Operators preserve the payload unaltered.
- ▶ Aggregate: The aggregate Operator removes payloads from all input Tuples.
- ▶ Join and Barrier: These Operators concatenate the payloads of the input Tuples.

Creating payloads

Every generated OPortN_t class has the following methods to attach a payload to output Tuples:

```
void setPayload(const unsigned char* payload, const unsigned int size)
```

The setPayload method makes a copy of the payload data, so the memory pointed to by payload can be temporary and should be freed by the caller if dynamically allocated:

```
void adoptPayload(unsigned char* payload, const unsigned int size)
```

The adoptPayload method accepts a pointer to dynamically allocated memory, and does not copy it. The Tuple object is responsible for freeing the data when it is no longer needed, and the caller should not free it.

Accessing and removing payloads

Every generated IPortN_t class has the following methods to access attached payloads:

```
const unsigned char* getPayload(unsigned int& size)
```

This method returns a pointer to the payload for read-only access:

```
unsigned char* releasePayload(unsigned int& size)
```

This method returns a pointer to the payload, and permanently removes it from the Tuple. The caller is responsible for freeing the returned memory.

6.5.4 Reusing user-defined Operators

The Fir example presented in this section is designed to be customized with the coefficient option and there may be multiple Instances of the Operator in the application. However, as long as the module name given is exactly the same, there will only be one set of generated files and the same implementation will be used for each Instance. For example:

```
stream Fir(Filtered : Integer)
    := Udop(Waveform)
```

```

["FirFilter"]
{coefficients="4,0,1,1,1,1,1,1,1,0,4"}

stream Fir2(Filtered2 : Integer)
:= Udop(Filtered)
["FirFilter"]
{coefficients="1,2,3,4,4,3,2,1"}

```

This application processes Tuples through two consecutive filters with a different set of coefficients, but only requires the single implementation of FirFilter. The application will create two Instances of the UDOP_FirFilter class, representing the two Operators, and the processCmdArgs method will be called with different arguments.

6.5.5 Type-generic Operators

The ability to reuse Operators would be limited if Operators were limited to operate on fixed types of input/output streams. The compiler's -L option instructs the compiler to generate Tuple classes with a reflective API, enabling the creation of "type-generic" Operators. A type-generic Operator uses the reflective API to determine stream types dynamically at run time.

When compiled with the -L option, each generated IPortN_t and OPortN_t class includes the following reflection API methods:

- ▶ `const hash_map<string,unsigned>& getAttributeNames() const`
Return a list of Attribute names as a hash_map.
- ▶ `const DpsValueHandle getAttributeValue(const string& attrb) const`
Obtains a read-only DpsValueHandle object for a given Attribute.
- ▶ `DpsValueHandle getAttributeValue(const string& attrb)`
Obtains a writable DpsValueHandle object for a given Attribute.
- ▶ `void assign(const ReflectiveTuple& otuple)`
Copies Attributes from the Tuple argument.

The DpsValueHandle class provides access to an Attributes value and type information. The Attributes type is obtained from a DpsValueHandle object with the getType method, which returns one of DPS::INTEGER, DPS::INTEGER_LIST, DPS::FLOAT, and so on. The value of an Attribute is obtained from the DpsValueHandle by casting the object to the appropriate type, or a string representation can be obtained via the toString method.

For example, a user-defined Sink Operator might output a generic identifier Attribute as follows:

```
DpsValueHandle value=ituple.getAttribute("id");

if (ituple.getType() == DPS::INTEGER) {
    int32_t const & ival = value;
    fprintf(outf,"%08X",ival);
} else {
    fprintf(outf,"%s",(const char *)value.toString());
}
```

As another example, consider an Operator that simply time stamps incoming Tuples of any type by adding a timestamp Attribute to incoming Tuples. The processInput0 method could be implemented as follows:

```
void UDOP_TimestampOp::processInput0(const IPort0_t & tuple)
{
    OPort0_t otuple;

    // copy attributes from tuple to otuple
    otuple.assign(tuple);

    // add timestamp attribute
    DpsValueHandle value = otuple.getAttribute("timestamp");
    int32_t & timestamp = value;
    timestamp = time(NULL);

    submit0(otuple);
}
```

Because this timestamp Operator uses the assign method to copy all Attributes, it could be reused many times with different streams types. For example,

```
stream Message(Timestamp : Integer, Subject : String, Body : String)
:= Udop(MessageIn)
["TimestampOp"]
{}

stream Alert(Timestamp : Integer, Description : String)
:= Udop(AlertIn)
["TimestampOp"]
{}
```

6.5.6 Port-generic Operators

In addition to making user-defined Operators generic with respect to input/output stream types, they can also be made generic with respect to the number of input/output streams. To create a port-generic Operator, you must add `#define DPS_PORT_GENERIC_UDOP` to the `UDOP_<module>_include.h` file of the Operator.

When `DPS_PORT_GENERIC_UDOP` is defined, the Operator API is changed as follows:

- ▶ `void processInput(const DpsTuple& tuple, unsigned iport)` replaces the usual `processInputN` methods and is responsible for processing incoming Tuples from all input streams. The `iport` parameter indicates the source of the incoming Tuple.
- ▶ `void submit(DpsTuple const& tuple, unsigned oport)` replaces the usual `submitN` methods with the `oport` parameter, which indicates the output stream to which to submit the Tuple.
- ▶ `DpsTuple* newOutputTuple(unsigned oport)` is available for creating output Tuples for the output stream indicated by the `oport` parameter.
- ▶ `int getNumberOfInputPorts(), int getNumberOfOutputPorts()` is available for determining the number of input/output streams.

Because one method is used to handle multiple streams, you cannot use the `IPortN_t` or `OPortN_t` classes. For port-generic Operators, you must also use the `-L` option to get the reflective API. However, to use the reflective API, you will also need to cast the Tuple argument of `processInput` and the return value of `newOutputTuple` to the `DpsReflectiveTuple` type using a dynamic cast. For example:

```
DpsReflectiveTuple const *ituple;  
ituple = dynamic_cast<DpsReflectiveTuple const *>(&tuple);
```

The following is an example `processInput` method for a port-generic Operator:

```
void UDOP_MergeOp::processInput(const DpsTuple& tuple, unsigned iport)  
{  
    DpsReflectiveTuple const *  
        ituple = dynamic_cast<DpsReflectiveTuple const *>(&tuple);  
    DpsReflectiveTuple *  
        otuple = dynamic_cast<DpsReflectiveTuple *>(newOutputTuple(0));  
  
    otuple->assign(*ituple);  
    DpsValueHandle value = otuple.getAttribute("channel");  
    int32_t & channel = value;  
    channel = iport;
```

```

    submit(*otuple, 0);
    delete otuple;
}

```

This Operator accepts any number of input streams and outputs all incoming Tuples on a single output stream, and adds the channel Attribute to indicate the source port number of the Tuple. Notice that the newOutputTuple method returns a dynamically allocated Tuple that needs to be deleted after it is submitted.

6.5.7 Multi-threaded Operators

Another form of the user-defined Operator is MTUdop, which defines a multi-threaded Operator. In contrast to the single-threaded UDOP, which outputs Tuples when Tuples are received (for example, in processInputN), the multi-threaded MTUdop has a separate thread running that can submit Tuples independently of receiving Tuples. For example, the MTUdop is often used to output Tuples at a fixed time interval. The overall data flow of a MTUdop is shown in Figure 6-2.

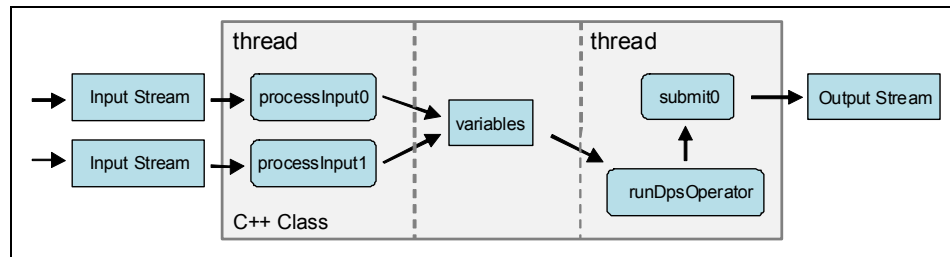


Figure 6-2 Overall flow of a MTUdop

The Operator's C++ class includes processInputN methods, such as the UDOP and an additional runDpsOperator. The runDpsOperator method is invoked from a separate thread after the Operator has been initialized. Because this method runs in a separate thread, it can submit output Tuples at any time independent of receiving Tuples. Because the runDpsOperator and the processInputN methods run in separate threads, if they access any shared member variables, then they must be synchronized to prevent access to the shared state at the same time.

Here is an example MTUdop that outputs statistics about an input stream at a fixed interval:

```
stream Beacon(Timestamp : Integer, MinV : Short, MaxV : Short, SumV :
Short)
    := MTUdop(Waveform)
    ["BeaconOp"]
    {}
```

The input stream Waveform has a single Attribute Value of type Short, and the BeaconOp Operator will generate the minimum, maximum, and sum of all values received thus far every second.

The same files are generated for this Operator as for the single-threaded Operators:

- ▶ UDOP_BeaconOp_includes.h
- ▶ UDOP_BeaconOp_members.h
- ▶ UDOP_BeaconOp.cpp

To implement the beacon, we declare a shared state as member variables in UDOP_BeaconOp_members.h:

```
// operator state
time_t startTime;
int16_t minValue,maxValue,sumValue;

// mutex to synchronize processInput0 and runDpsOperator
UTILS_NAMESPACE::Mutex runMutex;
```

The startTime variable is used to generate a time stamp on each record, and the minValue, maxValue, and sumValue maintain the current minimum, maximum, and sum for all Tuples received. Because processInput0 and runDpsOperator both access these variables, a mutex is declared to implement a critical region for each method. The UTILS_NAMESPACE::Mutex class is provided by the Streams environment and provides a convenient wrapper for a pthreads mutex.

The initialization method of this example sets the starting values of the Operator state as follows:

```
void UDOP_BeaconOp::initializeOperator()
{
    startTime=time(NULL);
    minValue=0x7fff;
    maxValue=0x8000;
    sumValue=0;
}
```

The processInput0 method simply updates the minValue, maxValue, and sumValue members each time a Tuple is received. This method is implemented as follows:

```
void UDOP_BeaconOp::processInput0(const IPort0_t & tuple)
{
    int16_t val=tuple.get_Value();

    AutoMutex am(runMutex); // acquire lock, released at end of block
    if (val<minValue)
        minValue=val;
    if (val>maxValue)
        maxValue=val;
    sumValue+=val;
}
```

The AutoMutex class is used to create a critical section that prevents the runDpsOperator thread from accessing the minValue, maxValue, and sumValue members simultaneously. The AutoMutex class constructor does not return until a lock is obtained on runMutex. The destructor releases the lock, which is invoked when the end of the am variable's scope is reached (that is, when the method returns).

The runDpsOperator method is as follows:

```
void * UDOP_BeaconOp::runDpsOperator(void * threadArgs)
{
    while (!getPEHandle().getShutdownRequested()) {
        {
            OPort0_t otuple;

            AutoMutex am(runMutex);
            otuple.set_Timestamp(time(NULL)-startTime);
            otuple.set_MinV(minValue);
            otuple.set_MaxV(maxValue);
            otuple.set_SumV(sumValue);
            submit0(otuple);
        }
        getPEHandle().blockUntilShutdownRequest(1);
    }

    return NULL;
}
```

The overall operation of this method is to continue looping as long the Operator is running, blocking for 1 second, and submitting an output Tuple each iteration.

The operation of the method is controlled by a built-in object returned by the `getPEHandle()`, which is available in every user-defined Operator class. The `getShutdownRequested` method returns true when the application is being shut down, and the `runDpsOperator` method should return at that point to allow the thread to terminate. The `blockUntilShutdownRequest` method takes a timeout argument, and blocks for that amount of time (seconds) or until the application shuts down.

Again, the `AutoMutex` class is used here to create a critical region around the code that accesses the shared state in `minValue`, `maxValue`, and `sumValue`. Because the *am* variable's scope is the nested code block, a new `AutoMutex` object is constructed and de-constructed for each iteration and before the call to `blockUntilShutdownRequest`. This allows the `processInput0` function to obtain a lock while the `runDpsOperator` thread is blocked in `blockUntilShutdownRequest`.

6.6 User-defined Built-in Operators

User-defined Built-in Operators (UBOPs) are similar to user-defined Operators, but offer a greater degree of customization and can be used by any Streams Application where they are installed. In many cases, a type-generic/port-generic Operator is flexible enough to meet the needs of a reusable Operator. However, there are a number of factors that make a UBOP more desirable or necessary:

- ▶ Broad deployment in many applications
- ▶ Syntax similar to Built-in Operators
- ▶ Window and aggregation support
- ▶ Output assignments
- ▶ Statically generated code (verses dynamic Attributes, types, and so on)

The benefits of the UBOP come at the cost of complexity, however. UBOPs are implemented as code generators, which are always more difficult to write and more error prone than implementing the code they generate directly. Furthermore, implementing a UBOP requires some knowledge of Perl in addition to C++. Because UBOPs must generate a C++ Operator implementation similar to the user-defined Operator, you should be familiar with implementing user-defined Operators before learning UBOPs.

6.6.1 Overview

UBOPs are registered with a local Streams installation, making them available to any Streams Application built from that installation. In order to provide some organization and avoid naming clashes, UBOPs are organized into toolkits. Each toolkit contains one or more related UBOPs in a single root directory. Each toolkit

has a reverse domain-name style name space that disambiguates Operators with the same name, for example, `com.superstore.inventory.analytics`.

Each UBOP consists of a set of template files used to generate code for the Operator and a syntax specification. The template files are standard C++ files with embedded Perl code. The syntax specification is an XML file describing restrictions about how the Operator may be used, for example, it supports at most one input stream.

Perform these steps to create a new UBOP:

1. Create the UDOP directory structure.
2. Generate starting templates.
3. Create a test application.
4. Modify the starting templates.
5. Create the syntax specification.

6.6.2 Create the UDOP directory structure

Each toolkit has a root directory containing one or more UBOPs. The UBOP is contained within a subdirectory of the toolkit directory using the full name of the Operator including name space, for example, `com.superstore.inventory.analytics.Aging`. The UBOP directory should contain a single subdirectory named `generic` that contains both the template files and syntax specification. A complete UBOP directory structure would look as follows:

```
InventoryToolkit/  
    com.superstore.inventory.analytics.Aging/  
        generic/  
            Aging_h.cgt  
            Aging_cpp.cgt  
            restriction.xml
```

6.6.3 Generate starting templates

The `spademkop.pl` script can be used to generate initial template files as a starting point. Simply run the `spademkop.pl` script from a UBOP's `generic` directory that does not contain any template files to have it generate basic templates for you.

When you run `spademkop.pl` or compile an application that uses the UBOP, the template files will be translated into Perl modules in the same directory with the `.pm` extension. These Perl modules are used to compile applications and should be left untouched.

6.6.4 Create a test application

During the creation of your UBOP, you want to invoke the Streams compiler on an example application to test the output generated from your templates. To create a test application, simply create a basic Streams Application that invokes your UBOP using the full name of the Operator. For example,

```
stream Combined(Id : String, Level : Integer,  
               Clamped : Integer, Channel : Integer)  
  := com.ibm.example.Intersect(Src1;Src2)  
  []  
  { Clamped := COND($1.Level>5,5,$1.Level) }
```

To use the Operator, you must also add the -t option to the **spadec** command in your Makefile, which indicates the location of the toolkit directory containing the Operator.

6.6.5 Modify the starting templates

Templates are used to generate both the header and C++ file that implement the Operator as a C++ class. The generated file is produced by copying the template file to the output, replacing certain portions of the template with the output of embedded Perl statements.

Perl code within the template is enclosed with the delimiters <% and %>. All text outside of those delimiters is output verbatim to the generated file. For example, the following statement would generate "Counting 1 2 3.", with the portion between the <% and %> replaced with the output from the enclosed Perl code:

```
Counting <%  
for my $i (1,2,3) {  
    print $i," ";  
}  
%>.
```

As a shortcut, the delimiters <%= and %> can be used to output the value of a Perl expression without having to use **print**. For example, you can output the value of a variable with <%= \$perlvariable %>.

Header templates

The header file template is named <operator>_h.cgt and contains the C++ class declaration for the Operator implementation. The spadecop.pl script generates all the necessary boilerplate, and you only need to add any member variables that your Operator requires to maintain state.

For example, at the end of the class declaration, you might add:

```
private:
    int tupleCount[<%= $context->getNumberOfInputPorts() %>];
```

This declares an array to keep track of the number of Tuples received from each input port. The `$context->getNumberOfInputPorts()` call returns the number of input ports. “Perl API” on page 293 provides more information about the Perl API available for generating code.

C++ templates

The C++ template is used to generate the C++ file that implements all the methods of the Operator's C++ class. Again, the `spademkop.pl` script generates all the necessary boilerplate when you create the starting templates. The two main sections that need to be modified are the sections that generate the `processN` and `process` methods.

For Operators with input ports, you will modify the section that generates one `processN` method for each input stream. Note that the methods must be named `processN` in UBOPs, whereas they are named `processInputN` in UDOPs. You will use the predefined Perl context object to get information about each input stream and generate a `processN` method for it.

The basic generation loop looks like the following code:

```
<%
    for (my $i=0; $i<$context->getNumberOfInputPorts(); ++$i) {
        my $istream = $context->getInputStreamAt($i);
void<%= $opname %>::process<%= $i %>(const <%= $istream->getType() %>&
tuple) {
        const<%= $istream->getType() %>&<%= $istream->getCppName() %> = tuple;
        // C++ code here
    }
    <%
}
%>
```

The text in bold is literal text that gets copied to the generated output, and the black text is Perl code that is executed to generate output. The loop uses the `getNumberOfInputPorts` method to generate one process method for each input stream. See “Perl API” on page 293 for a description of the other methods used in this example.

Here is example output from the basic loop described above:

```
void BIOP_Example::process0(const Src1_t & tuple) {
    const Src1_t& Src1_1 = tuple;
```

```

    // C++ code here
}

void BIOP_Example::process1(const Src2_t & tuple) {
    const Src2_t& Src2_2 = tuple;
    // C++ code here
}

```

The local variable declaration generated with the `getCppName` method is important because that is the name used in the C++ code generated for the Operator's output assignments. The C++ code for output assignments is generated by the Streams compiler, and if your Operator supports them, then you would use the `getAssignment` method described below to output it in your template.

Of course, the basic loop above generates an empty function. The example UBOP section presents a real example demonstrating how the process method gets fleshed out with actual code.

Perl API

The following is a partial list of the Perl objects and their methods available to templates for generating code. Refer to the language's reference manual for a complete list of objects and methods available.

The context object

This object provides access to all the information available about the Operator being generated. For example:

- ▶ `getNumberOfInputPorts`: Returns the number of input ports.
- ▶ `getInputStreamAt`: Returns the input stream object describing the specified input port.
- ▶ `getInputStreams`: Returns the collection of input stream objects for all input ports.
- ▶ `getNumberOfOutputPorts`: Returns the number of output ports.
- ▶ `getOutputStreamAt`: Returns the output stream object describing the specified output port.
- ▶ `getOutputStreams`: Returns the collection of output stream objects for all output ports.

The input stream object

The input stream object provides information about an input stream of the Operator being generated. Input stream objects are obtained from the context object. For example:

- ▶ `getName`: Returns the name of the input stream.
- ▶ `getType`: Returns the C++ type name used to represent the stream's Tuples.
- ▶ `getPortIndex`: Returns the port index of the stream.
- ▶ `getCppName`: Returns the name used in C++ expressions that refer to Attributes of the stream.
- ▶ `getNumberOfAttributes`: Returns the number of objects in the stream's Attributes collection.
- ▶ `getAttributeA`: Returns an Attribute object from the stream's Attributes collection at the given position.
- ▶ `getAttributeByName`: Returns the Attribute object from the stream's Attribute collection with the given name.
- ▶ `getAttributes`: Returns the stream's Attributes collection.

The output stream object

The output stream object provides information about an output stream of the Operator being generated. Output stream objects are obtained from the context object. For example:

- ▶ `getName`: Returns the name of the output stream.
- ▶ `getType`: Returns the C++ type name used to represent the stream's Tuples.
- ▶ `getPortIndex`: Returns the port index of the stream.
- ▶ `getCppName`: Returns the name used in C++ expressions that refer to Attributes of the stream.
- ▶ `getNumberOfAttributes`: Returns the number of objects in the stream's Attributes collection.
- ▶ `getAttributeAt`: Returns an Attribute object from the stream's Attributes collection at the given position.
- ▶ `getAttributeByName`: Returns the Attribute object from the stream's Attribute collection with the given name.
- ▶ `getAttributes`: Returns the stream's Attributes collection.

The Attribute object

The Attribute object represents one Attribute of the Tuple type associated with an input or output stream. Attribute objects are obtained from an input or output stream object. For example:

- ▶ `getName`: Returns the name of the Attribute.
- ▶ `getType`: Returns the C++ type name used to represent values of the Attribute.
- ▶ `hasAssignment`: Returns whether there is an assignment made to this Attribute.
- ▶ `getAssignment`: Returns the C++ expression for the Attribute's assignment.
- ▶ `hasAssignmentAggregate`: Returns true if there is an aggregate over this Attribute's assignment.
- ▶ `getAssignmentAggregate`: Returns the assignment aggregate function name, such as Min, Max, and Avg.

6.6.6 An example UBOP

In this section, we present an example UBOP Operator that performs the intersection of multiple input streams into a single output stream that contains the Attributes that are defined in every input stream and discards the rest. The Operator also supports the addition of new Attributes that are defined by some expression of the other output Attributes using an output Attribute assignment.

The test program is:

```
[Application]
test
```

```
[Program]
stream Src1(
    Id : String,
    Level : Integer,
    Quality : Float
)
:= Source()["file:///src1_in.dat",noDelays, csvFormat]
{1,2,3}

stream Src2(
    Id : String,
    Level : Integer,
    Location : String,
    Priority : Integer
```

```

)
:= Source()["file:///src2_in.dat",noDelays,csvFormat]
{1,2,3,4}

```

```

stream Combined(Id : String, Level : Integer, Clamped : Integer,
Channel : Integer)
:= com.ibm.example.Intersect(Src1;Src2)
[]
{ Clamped := COND($1.Level>5,5,$1.Level) }

```

The input streams Src1 and Src2 have two Attributes in common (Id, Level), so they are passed on to the output. Clamped is a new Attribute that is defined by an expression of the common Attributes. Channel is a distinguished name that can be used to include the input port number that is the source of the output Tuple.

Note that the output assignment refers to the Level Attribute of stream 1 even though output Tuples are generated from input Tuples on both stream 1 and stream 2. As a convention, stream 1 is used to refer to any Attributes in output Attribute assignments, but because these expressions may only use Attributes that are common to all input streams, we can compute that expression for Tuples received from any of the input streams. Ideally, the expression could be written using simply 'Level' without any port indication, but the Streams compiler does not allow that to happen.

The Intersect header template

Because this Operator does not maintain any state, it does not require any member variables. The starting template generated by spademkop.pl script is used unmodified.

The Intersect C++ template

The complete code for generating the Operator's processN methods is:

```

<%
# calculate common attributes
my %names;
for(my $i=0; $i<$context->getNumberOfInputPorts(); ++$i) {
  my $istream = $context->getInputStreamAt($i);
  foreach my $attribute (@{$istream->getAttributes()}) {
    my $name=$attribute->getName();
    if (defined $names{$name}) {
      $names{$name}++;
    } else {
      $names{$name} = 1;
    }
  }
}

```

```

    }
  }
}

# generate process methods
my $cppName = $context->getInputStreamAt(0)->getCppName();
for(my $i=0; $i<$context->getNumberOfInputPorts(); ++$i) {
    my $istream = $context->getInputStreamAt($i);
    my $localName = $istream->getCppName(); %>
void<%= $opername%>::process<%= $i%>(const <%= $istream->getType()%>&
tuple) {
    const<%= $istream->getType()%>&<%= $localName%> = tuple;
    <%
    my $j=0;
    foreach my $ostream (@{$context->getOutputStreams()}) {
        print $ostream->getType() , " otuple${j};\n";
# generate output of common attributes
        foreach my $name(keys %names) {
            if ($names{$name} == $context->getNumberOfInputPorts()) {
                print "otuple${j}.set_${name}(tuple._${name});\n";
            }
        }
# generate output of derived attributes and special channel attribute
        foreach my $attribute (@{$ostream->getAttributes()}) {
            my $name=$attribute->getName();
            if (!defined $names{$name}) {
                if ($name =~ /Channel/) {
                    print "otuple${j}.set_${name}(${i});\n";
                } else {
                    my $rhs = $attribute->getAssignment();
                    $rhs =~ s/$cppName/$localName/g;
                    print "otuple${j}.set_${name}(${rhs});\n";
                }
            }
        }
    }
}

# generate submit
print "submit${j}(otuple${j});\n";
$j++;
}
%>
}
<% }%>

```

Look at the code under the calculate common attributes comment. This code uses the Perl API to traverse all the Operator's input streams and count how

many streams have Attributes of a certain name using the names array. We can then determine which Attributes are common to all streams by checking that the count in the names array is equal to the number of input streams.

Under the `generate process methods` comment, we have entered the basic loop described in “C++ templates” on page 292 with code to generate a complete method body for each input stream. The `Intersect Operator` only supports one output, but we use a `for each` loop to loop over each output stream to demonstrate how it would be done in general.

This Operator outputs a Tuple for every Tuple received on any input stream that contains all the common Attributes of the input Tuple, the derived Attributes, and, optionally, the channel Attribute. Under the `generate output of common attributes` comment, the loop looks at every Attribute counted earlier and copies every Attribute that is common to all input streams (that is, which count in `names` is equal to the number of input streams) to the output Tuple.

Under the `generate output of derived attributes and special channel attribute` comment, we loop through all the Attributes of the output stream and, for any Attribute that is not a common Attribute, determine if it is the special channel Attribute or a derived Attribute.

Derived Attributes are calculated from the Operator's output assignments. Remember from the introduction to this example Operator that the output assignments refer to Attributes of stream 1, even though it is used to calculate the Attribute for input Tuples from any port. We need to deal with that situation now to generate correct code.

The `getAssignment` method is used to get the C++ code that implements an output assignment expression. Because the output assignment refers to Attributes of stream 1 and we may be generating code for another port, we need to replace references to stream 1 with references to the stream for which we are currently generating code. We accomplish this task by using a simple string replace statement, `:$rhs =~ s/$cppName/$localName/g`. The `cppName` variable is the name used to reference Tuples of stream 1, and `localName` is the name used to reference Tuples of the stream currently being generated.

Finally, we must generate a call to the `submit` method to actually output the Tuple. The resulting methods generated by this template for the test program shown above are:

```
void BIOP_Combined::process0(const Src1_t & tuple) {
    const Src1_t& Src1_1 = tuple;
    Combined_t otuple0;
    otuple0.set_Level(tuple._Level);
    otuple0.set_Id(tuple._Id);
```

```

    otuple0.set_Clamped(DPS::COND(Src1_1._Level>static_cast<int32_t>(5),s
    tatic_cast<int32_t>(5),Src1_1._Level));
    otuple0.set_Channel(0);
    submit0(otuple0);
}

void BIOP_Combined::process1(const Src2_t & tuple) {
    const Src2_t& Src2_2 = tuple;
    Combined_t otuple0;
    otuple0.set_Level(tuple._Level);
    otuple0.set_Id(tuple._Id);
    otuple0.set_Clamped(DPS::COND(Src2_2._Level>static_cast<int32_t>(5),s
    tatic_cast<int32_t>(5),Src2_2._Level));
    otuple0.set_Channel(1);
    submit0(otuple0);
}

```

6.6.7 Syntax specification

The syntax specification defines the valid syntax for the UBOP. The syntax specification is given by the `restriction.xml` file in the UBOP directory. Streams provides an interactive script to generate the `.xml` file.

To generate the `.xml` file, simply run the `spadecfop.pl` script. The script will ask a series of fairly straightforward questions and then generate the `.xml` file.

6.7 Hardware acceleration

The Built-in Operators, along with the Streams runtime platform, offer the potential for large-scale parallel performance increases using commodity CPU clusters. This kind of task-level parallelism is useful if you have many Operators in the application. However, with the flexibility of user-defined Operators, you can also use other kinds of parallelism using Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs). Both GPUs and FPGAs are available as plug-in boards for commodity computers and provide acceleration using fine-grain parallelism.

GPUs are generally well suited for data-parallel algorithms, in which the same operations must be performed on many data elements. GPUs excel in algorithms that require large numbers of parallel computations, for example, floating point matrix math. FPGAs are best suited for algorithms that have less uniformity in the computations being performed, or that can be most efficiently computed using nonstandard operations and bit widths. Examples of such algorithms

include network security and cryptography, genomics, pattern matching, and many types of signal processing.

In the context of stream processing, there are two main categories of applications for which hardware acceleration can provide substantial benefits: applications with high-volume data sources, and applications that perform compute-intensive algorithms on individual data items (that is, within a single Operator).

Applications that process high-volume data sources can benefit from acceleration by filtering or aggregating data from the source, thus reducing the amount of data handled downstream from the source. FPGAs are well suited for this type of acceleration. For example, an FPGA might be used in an intrusion detection system to inspect every packet, at line speed, thus providing the fast-path behavior of initial detection and screening while passing packets that require more complex behavior downstream. In this way, a small core of the algorithm is handled at high speed on the FPGA and only a fraction of the traffic would need to be handled by the downstream processor for the more complex behavior.

Applications that perform heavy computation on individual items in a data stream are also good candidates for hardware acceleration. This includes applications that make use of signal processing algorithms, such as FIR and FFT, or image processing algorithms, such as object tracking, data clustering, and so on. In these applications, each item in a stream requires significant compute time on a conventional processor, but can be accelerated greatly in FPGAs, using both data parallel computing and process-level hardware pipelining, or GPUs for uniform floating-point matrix computations.

6.7.1 Integrating hardware accelerators

Programming tools are available for both GPUs and FPGAs. In both cases, these tools require some level of hand-optimization and a certain level of skill on the part of the programmer. Nonetheless, compiler technologies available today make it possible for software algorithm developers to make effective use of GPUs and FPGAs, by allowing the use of familiar programming languages, including C. In the case of GPUs, OpenCL and CUDA provide C-like languages for programming GPUs. For FPGAs, there are several C language compiler tools available, including Impulse C from Impulse Accelerated Technologies and Catapult C from Mentor Graphics.

The Impulse C-to-FPGA compiler can be used alongside Streams. An integration tool, mkaccel, is available that allows use of the Impulse C development tools to create FPGA modules that implement a Streams Operator and execute on FPGA computing boards available from Pico Computing, Inc. Impulse C integrates

nicely with Streams in part because, like Streams, Impulse C is based on a streaming programming model. Unlike other Stream computing nodes, however, Impulse C nodes can exploit the extensive instruction-level and process-level parallelism available on an FPGA.

How does this work? Invoking the mkaccel integration tool on a Streams program duplicates the Streams Project directory, and replaces a named stream by a user-defined Operator that contains all the code to communicate with FPGA board. The tool generates a template C file for the FPGA program, and generates the necessary makefiles. The FPGA program is implemented in standard C, meaning that FPGA algorithm development can be thought of as a software programming problem. A side benefit of this is that widely available C language debugging tools can be used to speed development.

Here is an example template, in this case, a FIR filter. This template has been generated to replace an Operator whose input Tuples have one Attribute (Value: Short) and output Tuples have one Attribute (Filtered: Short):

```
void FIR_pe(co_stream ValueStream,co_stream FilteredStream) {
    short Value;
    short Filtered;
    co_stream_open(ValueStream,0_RDONLY,INT_TYPE(16));
    co_stream_open(FilteredStream,0_WRONLY,INT_TYPE(16));

    while (1) {
        #pragma CO PIPELINE

        co_stream_read(ValueStream,&Value,sizeof(Value));
        co_par_break();

        /* COMPUTE OUTPUTS HERE */

        co_stream_write(FilteredStream,&Filtered,sizeof(Filtered));
    }
}
```

The template file contains all the boilerplate code necessary to input and output data as declared by the original FIR Operator. To complete the FPGA program, you simply insert standard C code after the comment to compute the output from the input.

6.7.2 Line-speed processing

By using FPGA acceleration, some applications can process incoming network data at line speed. The nature of FPGAs also makes it easier to determine the latency and maximum number of cycles required by an operation. This is due to the lack of an operating system or runtime instruction scheduling. Such predictability is extremely important in low-latency networking applications.

To cite one example, the Impulse C tools have been used to develop an example that decodes and filters FAST/OPRA financial data feed packets at 100 Mbps. This example makes use of an FPGA board from Pico Computing, which is equipped with an Ethernet port for direct packet processing.

The FAST/OPRA example decodes and filters packets progressively, byte-by-byte as the packet is received. Within a few cycles of receiving the last byte, the filter decision has been made and the fully decoded message is ready to send to the CPU for downstream processing. For demonstration purposes, the example algorithm filters incoming packets based on the security's symbol, but more interesting filters are also possible and can be coded using C language programming methods.

Because the C language is used to program the FPGA, it does not take long for a software algorithm developer to learn FPGA programming. Note, however, that the C code written by a software programmer may need to be optimized for parallel operation to achieve maximum performance. Still, the FAST/OPRA example required only a couple of days to implement from scratch using the currently-available tools, and using specifications for the FAST/OPRA feeds.

The logo for IBM InfoSphere Streams, featuring a stylized globe with a circular arrow around it, symbolizing data flow and processing.

IBM InfoSphere Streams installation and configuration

In this appendix, we detail the install and initial configuration of the IBM InfoSphere Streams (Streams) software product, both the runtime and developer's work bench, to the extent necessary to replicate most or all of the examples in this book.

A larger and more complete installation and configuration guide is the *Streams Installation and Administration Guide*.

Physical installation components of Streams

When installing Streams, there are two physical components to install, which are:

- ▶ The Streams runtime platform

If you are installing in a multi-node Streams runtime configuration, this part of the Streams installation would be repeated for each management node, application node, or mixed-use node.

- ▶ The Streams Studio developer's workbench

This part of the Streams installation would be repeated for each developer workstation that use Streams Applications.

Both of the above items are included on the same Streams installation media, along with the Streams product documentation, sample applications, and more. Both of the above items can be installed on the same tier, for example, a stand-alone workstation.

In addition to these two physical installation components, there are also two other physical installation components, which are:

- ▶ On each operating system server tier, there are a number of operating system support packages that must be installed prior to installing Streams.

Generally a base install of an operating system will not include these packages, although these packages are found on that operating system's install media.

A prerequisite checker program that comes with Streams lists the presence of each required package and whether the installed version is of the correct version. Missing or incorrect packages are listed by their operating system installation name.

It is easiest to "mount" the given operating system's installation medium, and install missing or incorrect packages via that operating system's package manager.

Note: In addition to the currently 10 or more required operating system supplied packages, the Streams installation media includes three or so additional operating system packages that are also required.

While these additional packages come with the Streams' installation media, they are installed manually before installing Streams, in the same manner used to install any other operating system packages.

- The Eclipse open source developer's workbench

Eclipse is an open source and extensible developers workbench that provides a robust framework for anyone wishing to provide developer's tools. For example, if you wrote a FORTRAN to Java source code converter, you could make that program available as an Eclipse plug-in, and use all of the pre-existing Eclipse source code managers, editors, and so on, and concentrate more on the specific functionality on which you want to focus.

Similarly, the Streams Studio program exists as a number of Eclipse plug-ins, and thus requires that Eclipse be installed prior to installing Streams Studio.

Installing Streams on single-tier development systems

In this section, we provide details about how to install the Streams runtime and developer's workbench on a single-tier, developers workstation type system. Installing Streams on this type of system satisfies the requirements of most of the examples found in this book.

Perform the following steps on each client and server tier:

1. Download and the unpack the Streams installation media.

Download and the unpack the Streams installation media. You must run a 32 bit version of Streams on a 32 bit operating system, and a 64 bit version of Streams on a 64 bit operating system. Generally this media file arrives as a "gunzip" with an embedded "tar" file.

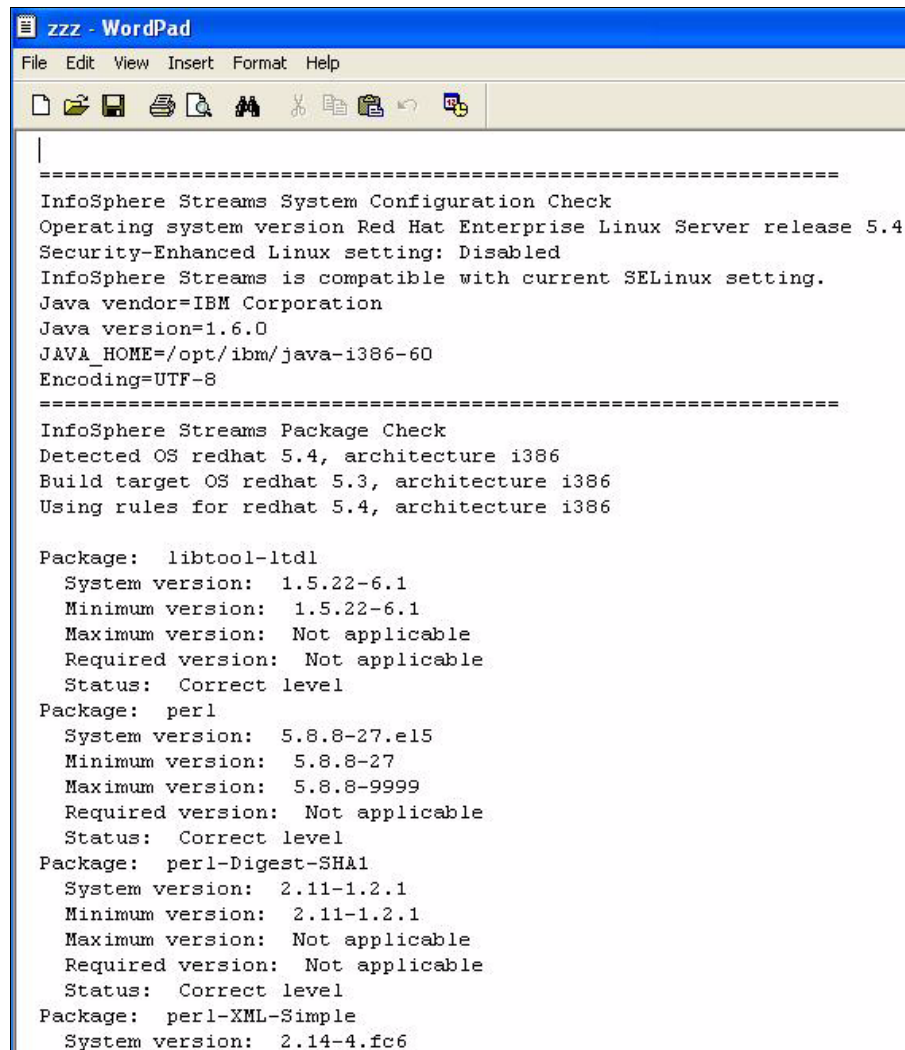
2. Read the *Streams Installation and Administration Guide*.

In the parent directory where the Streams software was unzipped is the *Streams Installation and Administration Guide*. Read that guide to gain a complete understanding of Streams software installation procedures and requirements.

3. Run the prerequisites checker.

In the parent directory where the Streams software was uncompressed is an executable program file named `dependency_checker.sh`. This program is the Streams software installation prerequisites checker.

Figure A-1 displays the output from this program. This program is non-destructive, which means it only outputs diagnostic information and makes zero changes to any settings or conditions.



```
=====
InfoSphere Streams System Configuration Check
Operating system version Red Hat Enterprise Linux Server release 5.4
Security-Enhanced Linux setting: Disabled
InfoSphere Streams is compatible with current SELinux setting.
Java vendor=IBM Corporation
Java version=1.6.0
JAVA_HOME=/opt/ibm/java-1386-60
Encoding=UTF-8
=====

InfoSphere Streams Package Check
Detected OS redhat 5.4, architecture i386
Build target OS redhat 5.3, architecture i386
Using rules for redhat 5.4, architecture i386

Package:  libtool-ltdl
  System version:  1.5.22-6.1
  Minimum version: 1.5.22-6.1
  Maximum version: Not applicable
  Required version: Not applicable
  Status:  Correct level
Package:  perl
  System version:  5.8.8-27.el5
  Minimum version: 5.8.8-27
  Maximum version: 5.8.8-9999
  Required version: Not applicable
  Status:  Correct level
Package:  perl-Digest-SHA1
  System version:  2.11-1.2.1
  Minimum version: 2.11-1.2.1
  Maximum version: Not applicable
  Required version: Not applicable
  Status:  Correct level
Package:  perl-XML-Simple
  System version:  2.14-4.fc6
```

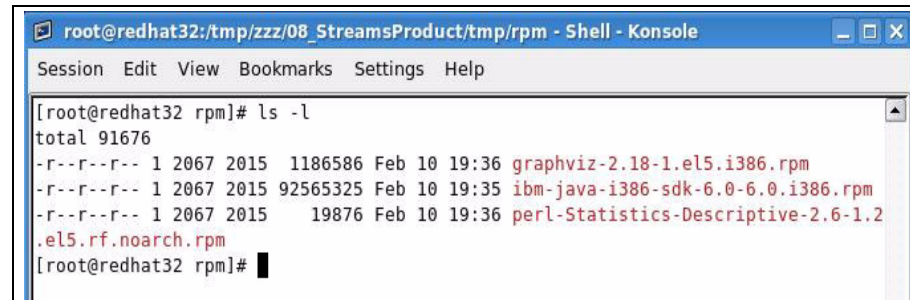
Figure A-1 Sample output of Streams prerequisites checker

4. Install operating system support packages and set variables.

The diagnostic output from the prerequisites checker program displayed in Figure A-1 serves as an outline of the required operating system packages that you need to install prior to installing Streams.

Most of these packages are located on the given operating system installation medium. These packages are identified in Figure A-1 on page 306 by the lines entitled, “Package”, and where the associated “Status” line reports a “Missing” or “Down-version.” A Down-version means, for example, that you need version 3 of a given Package, but version 2 is currently installed.

At least three of these packages are located on the Streams installation medium. From the parent directory where the Streams software was unzipped, these Streams supplied operating systems packages are located in the subdirectory named rpm. An example is shown in Figure A-2.



```
root@redhat32:/tmp/zzz/08_StreamsProduct/tmp/rpm - Shell - Konsole
Session Edit View Bookmarks Settings Help

[root@redhat32 rpm]# ls -l
total 91676
-r--r--r-- 1 2067 2015 1186586 Feb 10 19:36 graphviz-2.18-1.el5.i386.rpm
-r--r--r-- 1 2067 2015 92565325 Feb 10 19:35 ibm-java-i386-sdk-6.0-6.0.i386.rpm
-r--r--r-- 1 2067 2015 19876 Feb 10 19:36 perl-Statistics-Descriptive-2.6-1.2
.el5.rf.noarch.rpm
[root@redhat32 rpm]#
```

Figure A-2 Additional operating system packages to install from IBM

In Figure A-2, install these three packages in the following order: Java, Perl, and Graphviz. And these three packages are best installed after any of the operating system supplied packages.

Lastly, set the operating system environment variables entitled JAVA_HOME and PATH.

- The default value for JAVA_HOME is similar to /opt/ibm/java-i386-60, depending on your exact operating system and version of Java.
- The existing value for PATH should be prepended with the value, \$JAVA_HOME/bin. For example:

```
export PATH=$JAVA_HOME/bin:$PATH
```

5. Create a ParserDetails.ini file.

As root, create a ParserDetails.ini file by running the following command:

```
perl -MXML::SAX -e "XML::SAX -> add_parser(q(XML::SAX::PurePerl)) ->
save_parsers()"
```

ParserDetails.ini is a file required by Streams, and is found in the /usr/lib/perl5/vendor_perl/5.8.8/XML/SAX/ directory by default.

6. Install Streams.

In the parent directory where the Streams software was uncompressed is an executable program file named `InfoSphereStreamsSetup.bin`. This program is the Streams software installation program, and is ready to be run at this point.

Figure A-3 shows the Streams installer program, at step 3 of 9. Step 3 of the installer program essentially reruns the prerequisite checker, and will not allow you to proceed if errors or deficiencies are found.

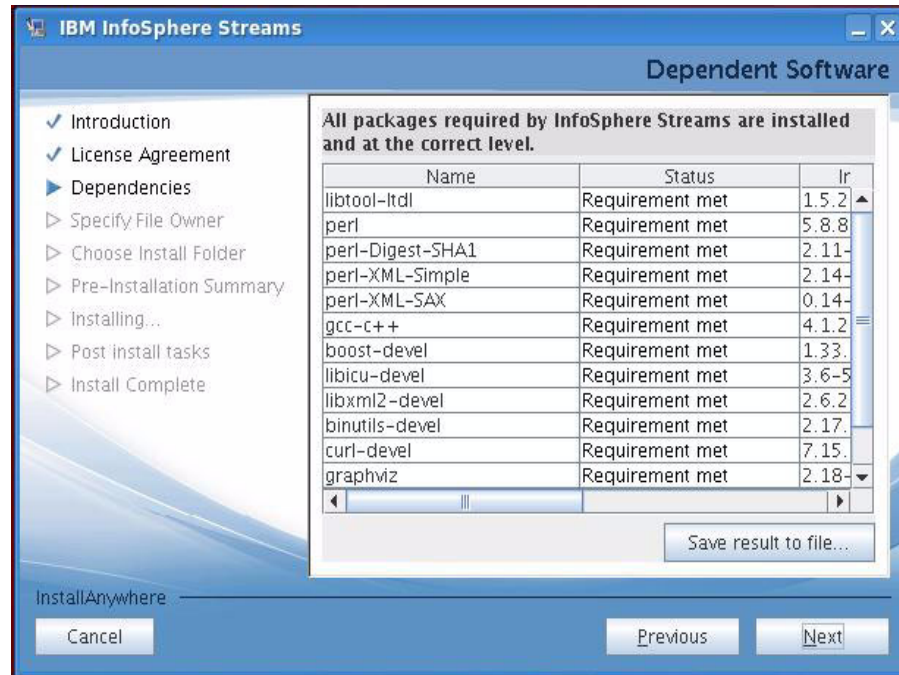


Figure A-3 Streams Installer program: Step 3

Generally, all of the default values used during the installation of Streams and while running the Streams installer program are adequate. Items specified during Streams installation, such as the default Streams user name, installation directory, and so on, can easily be overwritten or modified after the installation.

7. For each user of Streams, complete the following steps:
 - a. During the installation program execution above, a default Streams user name was supplied (the default user name is `streamsadmin`).

In the `streamsadmin` home directory is a file named `streamsprofile.sh`, which each user of Streams should source or copy.

- b. Each user of Streams also needs to enable encrypted and remote login by running the following sequence of commands:

```
cd
ssh-keygen -t dsa
chmod 600 .ssh
cd .ssh
cat id_dsa.pub >> authorized_keys
chmod 600 *
ssh localhost whoami
ssh { same hostname, not via network loopback } whoami
```

When executing the two **ssh** commands, answer Yes to any prompts to add Host or related.

8. Run **streamtool genkey**.

As the default Streams product user (streamsadmin), run the following command:

```
streamtool genkey
```

The command generates an SSH key pair for the Streams product.

At this point, you have installed the Streams runtime platform. To install the Streams Studio developer's workbench, perform the following steps:

1. Install Eclipse.

During the installation of the Streams runtime platform, Streams was installed in a given parent directory. The default directory is `/opt/ibm/InfoSphereStreams`. Under `/opt/ibm/InfoSphereStreams` is a child directory named `eclipse`.

The `eclipse` subdirectory contains the Streams' plug-ins for Eclipse.

As a product installation itself, Eclipse is unbundled, so you only have uncompress it into a given directory. Download the correct version and platform (including the correct bit version) of Eclipse and uncompress it into `/opt/ibm/InfoSphereStreams/eclipse`.

At this point, Eclipse is installed and co-located with the Streams' plug-ins. We have yet, however, to register these Streams' plug-ins with Eclipse, which we do by performing the following steps:

- a. Launch the Eclipse program by running the following command:

```
/opt/ibm/InfoSphereStreams/eclipse/eclipse
```

- b. Close any Welcome windows.

- c. From the Eclipse menu bar, select **Help** → **Install New Software**. Enter the following URL in the Work with field:

<http://download.eclipse.org/technology/imp/updates>

An example is shown in Figure A-4.

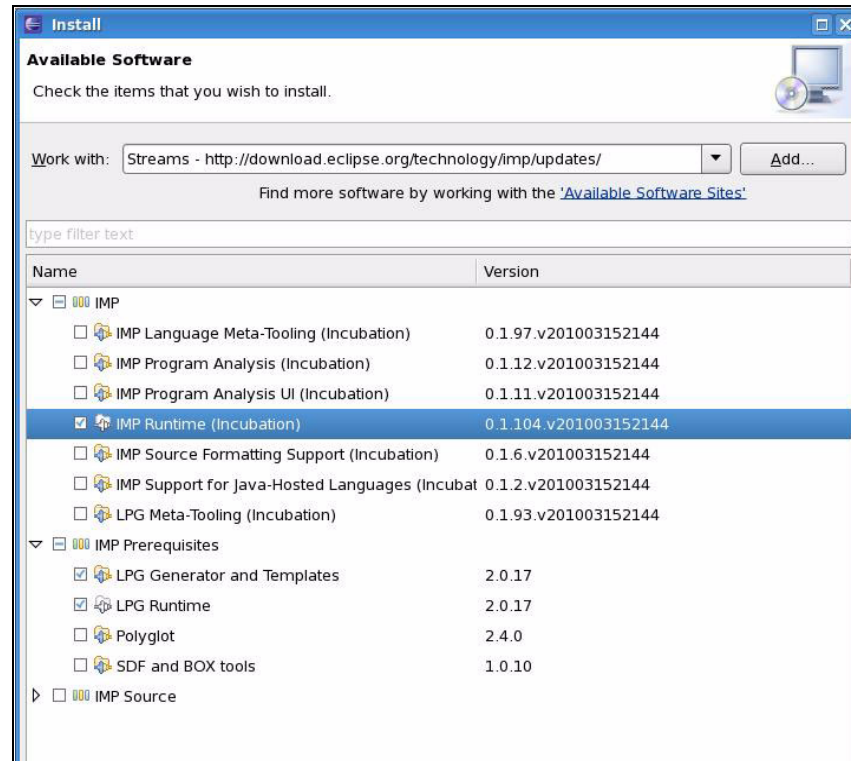


Figure A-4 Installing Eclipse updates: External dependencies

Check the **IMP Runtime (Incubation)** and **LPG Runtime and LPG Generator** check boxes. Click **Next**.

- d. Exit and re-enter Eclipse.

- e. From the Eclipse menu bar, select **Help** → **Install New Software**.

Click the **Add** button and then the **Local** button to navigate to /opt/ibm/InfoSphereStreams/eclipse, and select the four Streams plug-in packages to install. An example is shown in Figure A-5.

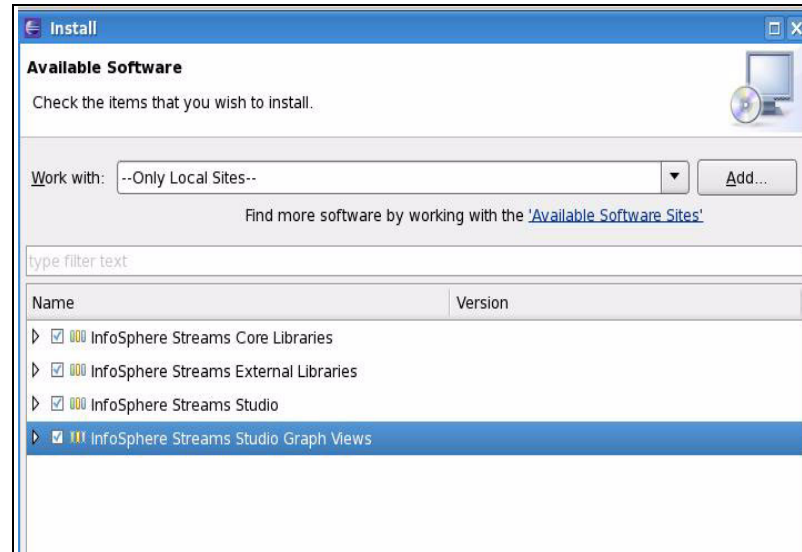


Figure A-5 Installing Streams plug-ins into Eclipse.

- f. Complete any navigation above to fully install the Streams plug-ins into Eclipse.
- g. Exit and re-enter Eclipse.
2. To verify the installation, try to create and run Example 2-2 on page 52 or the application shown in Figure 2-4 on page 46.



Toolkits and samples

In this appendix, we introduce the toolkits and samples that are included with the IBM InfoSphere Streams (Streams) platform. Although these components are important components of the base product, we chose to cover these important assets in a separate appendix because of their revolutionary nature. We describe the toolkit and sample assets available with IBM InfoSphere Streams V1.2, which is the current release of Streams at the time of the publication of this book. Because toolkits and sample assets may likely be added, changed, or possibly removed in future releases and even Fix Packs, we also describe how you can find the information for the toolkit and sample assets that are included in the particular release of the software you may be referencing.

In the previous chapters of this book, we discussed the valuable role the toolkit and sample assets play in enabling the delivery of high quality solutions using data in motion. We touch on that topic here, and also cover the following key questions and concepts.

- ▶ What is a toolkit or sample asset?
- ▶ Why do we provide toolkit or sample assets?
- ▶ Where do you find the toolkits and sample assets?
- ▶ What are the currently available toolkits and sample assets and how can you use them?

Overview

In this section, we discuss some of the basic information of the Streams toolkits and sample assets.

What is a toolkit or sample asset

Simply put, in the context of Streams, toolkits and samples are a collection of assets that facilitate the development of a solution for a particular industry or functionality. Derived from our experience in delivering streaming applications, these assets may be as simple as common Operators and adapters, or as complex as one or more sample applications, or anywhere in between. In general, sample assets tend to encompass the less complex of these assets, and are typically composed of simple Operators and adapters or a simple application flow. Toolkits tend to be more complex, with one or more complete sample applications. While this may typically be true, it is by no means a certainty. Development will deploy these helpful assets in the manner that seems most consistent with the way customers can relate to using them, based on our experience.

Why provide toolkit or sample assets

The book has covered at length the newness of this type of analysis for data in motion and analytical programming. Although you may be able to easily relate conceptually to how you might apply this new technology to the current challenges of your business or use it to move into compelling new areas of interest, when it comes time to move from the whiteboard to the keyboard, you may be feeling a bit overwhelmed.

The primary goal of the IBM InfoSphere Streams is not to get you to think about new ways of doing things that can make a difference, but is instead about enabling you to do those things. This components of the Streams platform are focused on enabling you to achieve that goal.

These assets are provided to give the new Streams developer working examples of what the language can do and how to program a working application to accomplish it. As a developer, you can use these assets:

- ▶ As a template to begin developing your own application
- ▶ To understand how to program an application in Streams
- ▶ To augment or include functionality into your Streams Application

- As an example of what type of applications can be developed in Streams and how they are structured

These assets do not come with any implicit warranties. They are not guaranteed to provide optimal performance for your specific environment, and performance tuning will likely still be required whether you use these assets alone or in conjunction with an application you have developed.

Where to find the toolkits and sample assets

The toolkits that are available with the product must be separately downloaded from the same site you used to download the software. Each is downloaded separately and the installation instructions for each of them is provided with them. There are also instructions in the product documentation.

The sample assets are downloaded with the product and placed in subdirectories under the sample directory of your Streams install directory. You set up the `$STREAMS_INSTALL` environment variable when you install the product. Each subdirectory will contain multiple sample assets and their supporting files (such as a makefile). Examples of the subdirectory listings for the sample assets are:

- `ls $STREAMS_INSTALL/sample/apps`
 - `algrtrade`
 - `fan_in_test`
 - `grep`
 - `loop_back_test`
 - `opra_edgeadapter`
 - `vmstat`
 - `wc`
 - `btree`
 - `fan_out_test`
 - `lois_edgeadapter`
 - `Makefile`
 - `regex`
 - `vwap`
- `ls $STREAMS_INSTALL/sample/atwork`
 - `aggregator`
 - `java_udop`
 - `partition`
 - `reflection`
 - `tumbling_window`
 - `apply1_apply2`
 - `join`
 - `perfcounter_query_api`
 - `split`

- udop_rss_feeds
- barrier
- list
- port_generic_udop
- stateful_functor
- udp_source_sink
- bcnt_and_normalize
- Makefile
- progressive_sliding_window
- stream_bundle
- user_defined_functions
- bytelist_tcp_udp_sources
- matrix
- punctor
- stream_monitor
- dynamic_profiles
- multiapp
- punctuations
- streams_profile
- dynamic_subscriptions
- multiple_output_ports
- punctuation_window
- tcp_source_sink
- import_export
- nodepool
- raw_udop_threaded_udop
- ls \$STREAMS_INSTALL/sample/demo
 - commodity_purchasing

Most often there is documentation, in the form of comments in the sample asset files, but it may not be consistent between the sample assets.

Note: These samples are examples of code to help you understand more about Streams and get started with the software. You should feel free to make a copy of any of them to use in your own applications and modify them as desired.

Currently available toolkit assets

In the current release of the software available at the time of this writing, there are two toolkits. These are:

- Streams Mining Toolkit
- Financial Markets Toolkit

As indicated by their names, one of these toolkits is focused on a vertical industry (financial markets), while the other is aligned with a functionality that could be used for many different industries (data mining). The components of these toolkits are provided to facilitate application development in or around each of these focuses. However, it is often possible to use some of the components in applications for other industries or even to build other functionality. Therefore, it is a good idea to review the toolkits and their documentation to see if there is any similarity to applications you may be developing and use them as examples.

As future toolkits become available, they will be available on the software download site where you obtained your software, for example, Passport Advantage®, and should appear in the release notes.

Streams Mining Toolkit

In this section, we discuss and describe the Streams Mining Toolkit.

Streams and data mining

Data mining has been a valuable analytical technique for decades. The process involves extracting relevant information or intelligence from large data sets based on such things as patterns of behavior or relationships between data or events in order to predict, react, or prevent future actions or behavior. To accumulate the amount of data needed to perform meaningful data mining has meant that most data mining has traditionally been performed on stored historical data.

For a certain class of problems, doing data mining analysis on historical data has limited value in being able to take certain specific actions, such as:

- ▶ Cyber security (requiring sub-millisecond reaction to potential network threats)
- ▶ Fraud detection
- ▶ Market trade monitoring
- ▶ Law enforcement

The challenge for these types of problems is to enable the techniques and algorithms used in data mining to be applied to real-time data.

This does not mean that the two types of analysis are mutually exclusive. A combined approach of applying the algorithms employed in your traditional mining of data at rest to streaming analysis of data in motion enables both proactive and reactive business intelligence. The Data Mining Toolkit is provided

to facilitate the development of applications to use your existing data mining analytics to be applied to real-time data streams.

Streams Mining Toolkit V1.2

This toolkit enables scoring of real-time data in a Streams Application. The goal is to be able to use the data mining models that you have been using against data at rest to do the same analysis against real-time data. The scoring in the Streams Mining Toolkit assumes, and uses, a predefined model. A variety of model types and scoring algorithms are supported. Models are presented using the Predictive Model Markup Language (PMML) standard for statistical and data mining models and an XML representation.

The toolkit provides four Streams Processing Language Operators to enable scoring:

- ▶ Classification
- ▶ Regression
- ▶ Clustering
- ▶ Associations

The toolkit also supports dynamic replacement of the PMML model used by an Operator to allow the application to be developed in a way to easily evolve to what is determined by the analysis.

How does it work

You start by building, testing, and training a model on a stored data set. After you have one or more models represented in PMML, which you created using modeling software that supports export of models as PMML (such as SPSS, Warehouse, SAS, and so on), you can incorporate the scoring based on this model into any Streams Application via the scoring Operators in Streams Processing Language. Specific Operators are compatible with specific types of models. Some of the ones supported include:

- ▶ For Classification models
 - Decision Tree
 - Logistic Regression
 - Naive Bayes
- ▶ For Regression models
 - Linear Regression
 - Polynomial Regression
 - Transform Regression
- ▶ For Clustering models
 - Demographic Clustering

- Kohonen Clustering
- For Associations models
 - Association Rules

At run time, real-time data is scored against the PMML model. The data streams flow into the scoring Operators and the results of the scoring flow out of the Operator as a stream. This process is shown in Figure B-1.

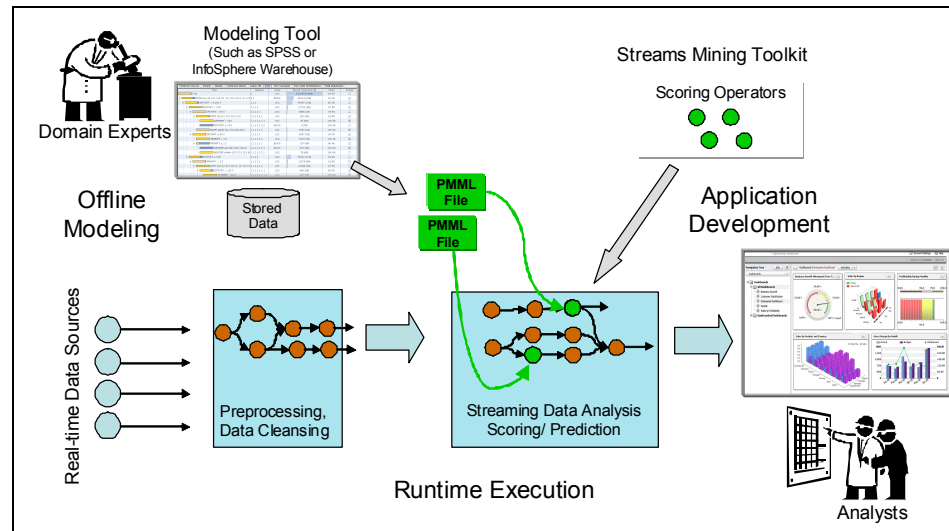


Figure B-1 Process flow of basic use case for Streams Mining Toolkit

By integrating the scoring of your models to process data in motion in real time, you can make decisions and take actions in time to make a significant difference. Your actions as a result of scoring real-time data will in turn change the data that is stored and used to build future models. This integration allows your analysis to evolve with improved business practices.

Financial Markets Toolkit

In this section, we discuss and describe the Streams Financial Markets Toolkit.

Why Streams for financial markets

Financial markets have long struggled with vast volumes of data and the need to make key decisions quickly. To address this situation, automated trading solutions have been discussed and designed in several forms. The challenge is to be able to use information from widely different sources (both from a speed

and format perspective) that may hold the definitive keys to making a better and profitable trades. Streams offers a platform that can accommodate the wide variety of source information and deliver decisions with the low latency that automated trading requires.

This combination of being able to use the variety of sources and deliver results in real time is not only attractive in financial markets, but may also provide insight into how to design an application for other industries that have similar needs.

Financial Markets Toolkit V1.2

This toolkit is focused on delivering examples and Operators that facilitate the development of applications to provide a competitive advantage to financial industry firms by using Streams. The examples provided demonstrate functionality that should be easy to integrate into their existing environment and reduce the time and effort in developing Streams-based financial domain applications. Our goal is to make it easy to use the unique strengths of Streams (real-time, complex analysis combined with low latency).

Because one of the core needs of financial markets is the wide variety of sources, one of the key focuses of this toolkit is to provide Source adapters for the more common market data feeds. It also provides adapters for some of the typical market data platforms and general messaging. These adapters make up the base layer of this toolkit's three layer organization, as shown in Figure B-2.

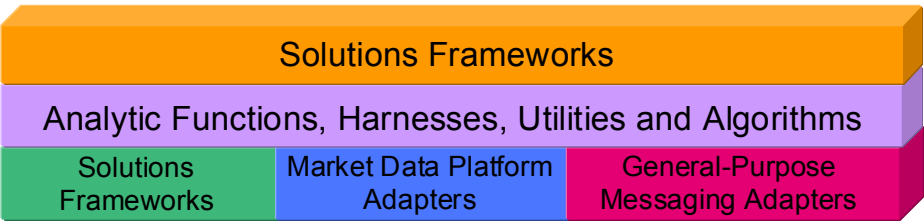


Figure B-2 Financial Markets toolkit organization

The components in the adapters' layer are used by top two layers of the organization and can also be used by your own applications. The functions' layer components are used by top layer and can also be used for your own applications. The components of the top layer represent the Solution Frameworks, starter applications that target a particular use case within the financial markets sector. These are typically modified or extended by your developers for your specific needs.

What is included in the Financial Markets Toolkit

The toolkit includes Operators to support adapters for Financial Information Exchange (FIX), such as:

- ▶ fixInitiator Operator
- ▶ fixAcceptor Operator
- ▶ FixMessageToStream Operator
- ▶ StreamToFixMessage Operator

It also supports WebSphere Front Office for Financial Markets (WFO) adapters with the following Operators:

- ▶ WFOSource Operator
- ▶ WFOSink Operator

For messaging, the toolkit includes an Operator to support the WebSphere MQ Low-Latency Messaging (LLM) adapter MQRmmSink Operator.

In the Function layer, this toolkit includes analytic functions and Operators, such as:

- ▶ Analytical Functions:
 - Coefficient of Correlation
 - “The Greeks”
- ▶ Put and Call values:
 - Delta
 - Theta
 - Rho
 - Charm
 - DualDelta
- ▶ Operators (UDOPs):
 - Wrapping QuantLib financial analytics open source package
 - Provides Operators to compute theoretical value of an option:
 - EuropeanOptionValue Operator (Provides access to 11 different analytic pricing engines, such as Black Scholes, Integral, Finite Differences, Binomial, Monte Carlo, and so on)
 - AmericanOptionValue Operator (Provides access to 11 different analytic pricing engines, such as Barone Adesi Whaley, Bjerksund Stensland, Additive Equiprobabilities, and so on)

Finally, the Solution Framework layer provides two extensive example applications:

- ▶ Equities Trading
- ▶ Options Trading

These examples are typically called *whitebox* applications, because customers can, and typically will, modify and extend them. These example applications are modular in design with plug replaceable components that you can extend or replace with your own. This modular design allows these applications to demonstrate how trading strategies may be swapped out at run time, without stopping the rest of the application.

The Equities Trading starter application includes a TradingStrategy module that looks for opportunities that have specific quality values and trends, the OpportunityFinder module looks for opportunities and computes quality metrics, and the SimpleVWAPCalculator module computes a running volume-weighted average price metric.

The Options Trading starter application includes the DataSources module, which consumes incoming data and formats and maps it for later use, the Pricing module, which computes theoretical put and call values, and the Decision module, which matches theoretical values against incoming market values to identify buying opportunities

These starter applications give the application developer good examples and a great foundation to start building their own applications.

Glossary

Access Control List (ACL). The list of principals that have explicit permission to publish, to subscribe to, and to request persistent delivery of a publication message against a topic in the topic tree. The ACLs define the implementation of topic-based security.

Analytic. An application or capability that performs some analysis on a set of data.

Application Programming Interface. An interface provided by a software product that enables programs to request services.

Asynchronous Messaging. A method of communication between programs in which a program places a message on a message queue, then proceeds with its own processing without waiting for a reply to its message.

Computer. A device that accepts information (in the form of digitalized data) and manipulates it for a result based on a program or sequence of instructions about how the data is to be processed.

Configuration. The collection of brokers, their execution groups, the message flows and sets that are assigned to them, and the topics and associated access control specifications.

Data Mining. A mode of data analysis that focuses on the discovery of new information, such as unknown facts, data relationships, or data patterns.

Deploy. Make operational the configuration and topology of the broker domain.

Engine. A program that performs a core or essential function for other programs. A database engine performs database functions on behalf of the database user programs.

Instance. A particular realization of a computer process. In regards to a database, the realization of a complete database environment.

Metadata. Typically called data (or information) about data. It describes or defines data elements.

Multi-Tasking. Operating system capability that allows multiple tasks to run concurrently while taking turns using the resources of the computer.

Multi-Threading. Operating system capability that enables multiple concurrent users to use the same program. This reduces the impact of initiating the program multiple times.

Optimization. The capability to enable a process to execute and perform in such a way as to maximize performance, minimize resource utilization, and minimize the process execution response time delivered to the user.

Process. An Instance of a program running in a computer.

Program. A specific set of ordered operations for a computer to perform.

Roll-up. Iterative analysis that explores facts at a higher level of summarization.

Server. A computer program that provides services to other computer programs (and their users) in the same or other computers. However, the computer that a server program runs in is also frequently referred to as a server.

Task. The basic unit of programming that an operating system controls. See also *Multi-Tasking*.

Thread. The placeholder information associated with a single use of a program that can handle multiple concurrent users. See also *Multi-Threading*.

Zettabyte. A trillion gigabytes.

Abbreviations and acronyms

AAS	Authentication and Authorization Server	JDK	Java Development Kit
CFG	Configuration	JDL	Job Description Language
CORBA	Common Object Request Broker Architecture	JE	Java Edition
CPU	Central Processing Unit	JMC	Job Manager Controller
DDY	Distributed Distillery	JMN	Job Manager
DFE	Distributed Front End	LAS	Logging and Auditing Subsystem
DFS	Data Fabric Server	LDAP	Lightweight Directory Access Protocol
DGM	Dataflow Graph Manager	Mb	Megabit
DIG	Dense Information Grinding	MB	Megabyte
DNA	Distillery Instance Node Agent	MNC	Master Node Controller
DPFS	General Parallel File System	NAM	Naming
DPS	Distributed Processing System	NAN	Nanoscheduler
DSF	Distillery Services Framework	NC	Node Controller
DSM	Data Source Manager	NDA	Network Data Analysis
DSS	Distillery Semantic Services	O/S	Operating System
DST	Distillery Base API	ODBC	Open DataBase Connectivity
EVT	Event Service	OPT	Optimizer
Gb	Gigabit	ORA	Ontology and Reference Application
GB	Gigabyte	OS	Operating System
GUI	Graphical User Interface	PE	Processing Element
I/O	Input/Output	PEC	Processing Element Containers
IBM	International Business Machines Corporation	PHI	Physical Infrastructure
IDE	Integrated Development Environment	PRV	Privacy
IKM	Information Knowledge Management	RAA®	Resource Adaptive Analysis
INQ	Inquiry Services	REF	Reference Application
ITSO	International Technical Support Organization	REX	Results/Evidence Explorer
JDBC	Java Database Connectivity	RFL	Resource Function Learner
		RMN	Resource Manager
		RPS	Repository

RTAP	Real-time Analytic Processing
SAGG	Statistics Aggregator
SAM	Streams Application Manager
SCH	Scheduler
SDO	Stream Data Object
SEC	Security
SMP	Symmetric MultiProcessing
SOA	Service-oriented Architecture
SODA	Scheduler
SPADE	Streams Processing Application Declarative Engine
SPC	Stream Processing Core
SRM	Stream Resource Manager
STG	Storage
Streams	IBM InfoSphere Streams
SWS	Streams Web Service
SYS	System Management
TAF	Text Analysis Framework
TEG	Traffic Engineering
TCP	Transmission Control Program
TCP/IP	TCP/Internet Protocol
TCP/UOP	TCP/User Datagram Protocol
TRC	Tracing (debug)
UBOP	User-defined Built-in Operator
UDF	User-defined Function
UDOP	User-defined Operator
UE	User Experience
UTL	Utilities
WLG	Workload Generator
URI	Uniform Resource Identifier
URL	Universal Record Locator
WWW	World Wide Web
XML	eXtensible Markup Language

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

Other publications

These publications are also relevant as further information sources:

- ▶ Kernighan, et al, *The C Programming Language*, Prentice Hall, 1988, ISBN 0131103628

Online resources

These websites are also relevant as further information sources:

- ▶ 2008 ACM SIGMOD International Conference on Management of Data, in Vancouver, Canada. Session 3: Streams, Conversations, and Verification
<http://portal.acm.org/citation.cfm?id=1376616.1376729>
- ▶ Ganglia Monitoring System
<http://ganglia.sourceforge.net/>
- ▶ LOFAR Outrigger in Scandinavia project
<http://www.lois-space.net/index.html>
- ▶ University of Ontario Institute of Technology Neonatal research project
<http://research.uoit.ca/assets/Default/Publications/Research%20Viewbook.pdf>

IBM Education Support

This section discusses IBM Education support for IBM InfoSphere Streams.

IBM Training

IBM Training enhances your skills and boosts your success with IBM software. IBM offers a complete portfolio of training options, including traditional classroom, private on site. and eLearning courses. Many of our classroom courses are part of the IBM “Guaranteed to run program”, which ensures that your course will never be canceled. We have a robust eLearning portfolio, including Instructor-Led Online (ILO) courses, Self Paced Virtual courses (SPVC), and traditional Web-based Training (WBT) courses. As a perfect complement to classroom training, our eLearning portfolio offers something for every need and every budget, and you only need to select the style that suits you.

Be sure to take advantage of our custom training plans to map your path to acquiring skills. Enjoy further savings when you purchase training at a discount with an IBM Education Pack online account, which provides a flexible and convenient way to pay, track, and manage your education expenses online.

The key education resources listed inTable 1 have been updated to reflect IBM InfoSphere Streams. Check your local Information Management Training website or chat with your training representative for the most recent training schedule.

Table 1 Education resources

Course Title	Classroom	Instructor-Led	Self Paced Virtual Classroom	Web-based Training
Programming for IBM InfoSphere Streams	DW321	3W721		

Descriptions of courses for IT professionals and managers are available at the following address:

http://www.ibm.com/services/learning/ites.wss/tp/en?pageType=tp_search

Visit <http://www.ibm.com/training> or call IBM training at 800-IBM-TEACH (426-8322) for scheduling and enrollment.

Information Management Software Services

When implementing an information management solution, it is critical to have an experienced team involved to ensure you achieve the results you want through a proven, low risk delivery approach. The Information Management Software Services team has the capabilities to meet your needs, and is ready to deliver

your Information Management solution in an efficient and cost-effective manner to accelerate your Return On Investment (ROI).

The Information Management Software Services team offers a broad range of planning, custom education, design engineering, implementation, and solution support services. Our consultants have vast technical knowledge, industry skills, and delivery experience from thousands of engagements worldwide. With each engagement, our objective is to provide you with a reduced risk and expedient means of achieving your project goals. Through repeatable services offerings, capabilities, and best practices using our proven methodologies for delivery, our team has been able to achieve these objectives and has demonstrated repeated success on a global basis.

The key Services resources listed in Table 2 are available for IBM InfoSphere Streams.

Table 2 Services resources

Information Management Services Offering	Short Description
IBM InfoSphere Streams Delivery Capability, found at the following address: http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/services/InfoSphere_Streams_Delivery_Capabilities.pdf	Our Information Management Software Services team has a number of capabilities that can support you with your deployment of our IBM InfoSphere Streams Solution, including: <ul style="list-style-type: none">► Installation and configuration support► Getting acquainted with IBM InfoSphere Streams► Mentored pilot design► Streams Programming Language education► Supplemental Streams programming support

For more information, visit our website at the following address:

<http://www.ibm.com/software/data/services>

IBM Software Accelerated Value Program

The IBM Software Accelerated Value program provides support assistance for issues that fall outside normal “break-fix” parameters addressed by the standard IBM Support contract, offering customers a proactive approach to support management and issue resolution assistance through assigned senior IBM Support experts who know your software and understand your business needs. Benefits of the IBM Software Accelerated Value Program include:

- Priority access to assistance and information
- Assigned support resources
- Fewer issues and faster issue resolution times
- Improved availability of mission-critical systems

- ▶ Problem avoidance through managed planning
- ▶ Quicker deployments
- ▶ Optimized use of in-house support staff

To learn more about IBM Software Accelerated Value Program, visit our website at the following address:

<http://www.ibm.com/software/data/support/acceleratedvalue/>

To talk to an expert, contact your local IBM Software Accelerated Value Program Sales Representative at the following address:

<http://www.ibm.com/software/support/acceleratedvalue/contactus.html>

Protect your software investment

To protect your software investment, ensure you renew your Software Subscription and Support. Complementing your software purchases, Software Subscription and Support gets you access to our world-class support community and product upgrades with every new license. Extend the value of your software solutions and transform your business to be smarter, more innovative, and cost-effective when you renew your Software Subscription and Support. Staying on top of on-time renewals ensures that you maintain uninterrupted access to innovative solutions that can make a real difference to your company's bottom line.

To learn more, visit our website at the following address:

<http://www.ibm.com/software/data/support/subscriptionandsupport>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this website:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

A

- AAS 129, 163, 167, 170, 179
- AAS service 159, 170
- Access Control List (ACL) 167
- ACL permissions 168
- Active Diagnostics 24
- adaptor toolkit 258
- adoptPayload method 282
- Aggregate Operator 205, 214
- alerting 15
- alerting pattern 104
- alerts 24
- alphanumeric source data 82
- analytic application 6
- analytic tools 13
- analytical rules 18
- anomalous readings 104
- application deployment 144
- Application Graph View 45
- Application Host 41
- application parameterization 229
- application profiling 183
- application programming interface (API) 84, 123
- application state 141
- astronomy 26
- astrophysics 27
- Attribute data type 58
- Attributes 44
- audit policy 163
- authentication module 163
- Authorization and Authentication Service (AAS) 129
- authorization policy 163, 167
- AutoMutex class 288

B

- Balanced Mode Scheduler 43
- Barrier Operator 205, 221
- BeaconOp Operator 287
- binary source data 82
- blogs 110, 118
- break point 252
- Built-in Adapters 48

- Built-in Operator 267, 272
- Built-in Sink Operator 272
- Built-in Source Operator 268
- Business Activity Monitoring 24
- business intelligence xii
- Business Logic Derived Events 24
- Business Service Management 24

C

- C++ 20
- C++ code 120, 258
- C++ templates 266
- C++ UDOP 123
- cellular phones 2
- Client/Server Era 10, 18
- coded algorithms 86
- collocate Operators 146
- Complex Event Processing 24
- complex events 25
- computing memory 2
- Confined Domains 44
- consolidation pattern 112
- continuous analysis 4
- continuous data streams 18
- continuous view 35
- Control application class 80
- control focused applications 80
- cosmic ray showers 26
- CrossOp class 281

D

- data acquisition 2, 6
- data entry 2
- data in motion 4, 13, 15–16
- Data Mining Toolkit 79, 95
- data model 36
- data persistence 84
- data reduction 136
- data sources 80
- data stream 23
- data warehousing xii, 14, 18
- database xii
- DB2 database 178, 258

- declarative computer language 40
- dedicating Operators 145
- Delay Operator 205, 220
- deployment implications 137
- design patterns 88
 - Alerting 88
 - Consolidation 88
 - Enrichment 88
 - Filter 88
 - Integration 88
 - Merge 88
 - Outlier 88
 - Parallel 88
 - Pipeline 88
 - Unstructured Data 88
- digital imaging 2
- digital radio observatory 27
- Discovery application class 80
- discovery focused applications 79
- discrete events 25
- Distributed Streams applications 47
- dominant bottleneck 155
- DpsValueHandle class 283
- dynamic applications 15
- dynamic data model 38
- dynamic Host selection 134

E

- early and late filtering 92
- eBooks 2
- Eclipse 20, 65, 248, 305
- Eclipse developers workbench 65
- Eclipse Perspective 66
- Eclipse plug-in 305
- Eclipse View 66
- Edgar database 20
- Edge adaptors 258
- Editor View 66
- EnrichingValues stream 107
- Enrichment pattern 106
- events of events 25
- eviction policy 201
- existing analytics 120
- export streams by name 139
- exporting streams 233

F

- feedback 15

- file based data access 81
- Filter pattern 89
- filtering
 - early and late 92
- finalization method 278
- Financial Markets Toolkit 316
- Financial Services applications 31
- financial services industry 30
- Fir12 function 266
- for-loops 230
- fraud detection 79
- FreeformText 111
- FunctionDebug 192
- Functor 142
- Functor history 214
- Functor Operator 57, 59, 204, 212

G

- gamma ray bursts 26
- Ganglia 151

H

- health care 28
- health monitoring 16
- Hello World 238
- hierarchical databases 14
- high availability 25
- high volume information 8
- High Volume Stream Analytics 24
- high volume streams 91
- historical data 5
- Host Controller (HC) 43, 128–129, 133, 145, 159
- HTTP/HTTPS connection 163

I

- IBM Informix Dynamic Server 258
- IBM InfoSphere Streams 1, 16, 23, 326
 - streamtool 72
- IBM InfoSphere Streams Console 72
- IBM InfoSphere Streams Studio 64, 152, 180
- IBM Research 17–18
- IBM solidDB 19, 48, 108
- IBM solidDB table query 261
- IBM WebSphere Front Office 19, 48, 258, 262
- Import and Export modifiers 47
- import streams by name 139
- importing streams 233

- incorporating existing analytics 120
- InetSource Operator 48, 263
- Information-Derived Events 24
- initialization method 278
- inject load 150
- input Attribute 59
- input Stream Operators 59
- Input Tuples 59
- instrumentation 13
- Integrated Development Environment 20
- Integration pattern 118
- intelligent cellular phones 9
- inter-arrival delay 210
- interconnected communication channels 4
- interconnectivity 3
- Internet of Things 12
- Internet Operator 258

J

- Java 20, 248
- Java UDOP 123
- Java visual debugger 248
- Join Operator 60, 63, 204, 222

K

- key performance indicators 24

L

- latency 144, 151
- Latency application class 80
- Latency focused applications 79
- Libdefs 192
- Lightweight Directory Access Protocol (LDAP) 165
 - authentication 156
 - parameters 166
- load balancing 25
- LOFAR Outrigger 27
- logical application design 76
- logical design 76
- LogParser 271
- Low Frequency Array 26
- low latency 79, 87
- lower volume applications 130

M

- Mainframe Era 10
- Management Host 41

- Mandatory Access Control (MAC) 172
- marked subgroups 61
- market surveillance 15
- marketing xii
- mathematical models 13
- medical and image scanners 4
- member Hosts 41
- Merge pattern 115
- mixed mode processing 231
- mixed mode programs 232
- Mixed-Use Host 41
- MP3 players 2
- multi-lingual sources 118
- multiple applications 138
- multiple Host Instance 132
- multi-threaded Operator 286
- multivariate monitoring 28

N

- Name Service 43
- natural language processing 86, 110
- network communication bandwidth 2
- network firewall 159
- news feeds 110
- Node Pools 44
- nodepools 145, 192
- non-traditional
 - data 7
 - data sources 4
 - information 8
 - sources 9

O

- ODBC enrichment Operator 108
- ODBCAppend Operator 261
- ODBCEnrich Operator 260
- ODBCSource Operator 260
- On Demand Era 11
- Online Analytic Processing 14
- Online Transaction Processing 14
- Open Source Eclipse 20
- operating system firewall 159
- Operator functionality segmenting 137
- Operator language syntax 207
- Operator parameters 200
- Operator workload 137
- Operators 204
- Operators in parallel 137

- optical scanning 2
- outlier detection 92
- Outlier pattern 92
 - classification 94
- output Attribute mapping 57

P

- Parallel pattern 96
- parallel processing 79, 109
- patient monitoring 15
- peak load period 87
- performance characteristics 149
- Performance Counter API 180
- Performance Counter interface 180
- performance improvement 148
- performance measurement 150
- performance problems 147
- performance targets 148
- performance testing 149
- Performance Visualizer windows 180
- personal data assistants 9
- physical component design 76
- Pipeline pattern 102
- pipelined processing 144
- pipelining 144
- Pluggable Authentication Module (PAM) 165
 - authentication 156
- Policy Modules 44
- policy modules 173
- port-generic Operator 285
- Prediction application class 80
- prediction focused applications 79
- Predictive Mode Scheduler 43
- Predictive Model Markup Language (PMML) 95, 318
- Predictive Processing 24
- preprocessor directives 202, 228
- process architecture 49
- Processing Element (PE) 49, 127, 129, 144, 152, 155–156, 171, 176, 246
 - confined 171
 - confined and vetted 171
 - fusing 155
- Processing Element Container (PEC) 42–43, 49, 129, 145, 156, 159
 - unconfined 171
- processor intensive Operators 145
- ProcessUnstructuredText 111

- programming model 39
- progressive key word 225
- Project Explorer View 69
- protocol support 210
- publish and subscribe 236
- Punctor 142
- Punctor Operator 205, 214, 220
- punctuation marks 196, 211

Q

- query optimizer 40

R

- radio astronomy 85
- radio telescopes 26
- RawInput stream 105
- RawText stream 110
- RawValues stream 94, 99, 107
- RBCompute hosts 133
- real time 2–3, 24, 26, 30
- real-time
 - analysis 2, 30
 - analytics 4, 8, 13
 - data 36
 - data streams 25
 - information 8
- real-time analytic processing (RTAP) 1, 15, 22
- recovery 175
- RecoveryMode 179
- Red Hat Enterprise Linux 65, 172
- Red Hat Linux 160
- Redbooks Web site 330
 - Contact us xv
- relational databases 14
- resource utilization 151
- RSA keys 160, 162–163, 167
- runtime architecture 127
- runtime cluster 19
- runtime environment 39

S

- SAM 128, 144, 159, 176, 179
- satellite images 35
- satellite photo 35
- scalability 144
- Scans stream 273
- schema 194

- Schema Definition 48, 58
- schema definition 195
- Schema Definition Attribute 56
- scoring 318
 - Associations 318
 - Classification 318
 - Clustering 318
 - Regression 318
- security and privacy 13
- security policy 163
- security policy template 163, 165
- segmenting streams 141
- Self Contained Streams Application 47
- SELinux 156, 160, 171
 - Enforcing mode 173
 - policy 160
 - security policies 172
- sensors 4
- Sink Operator 47–48, 83, 105, 188, 204
- sizing the environment 136
- Sliding windows 61, 198, 200, 216
- smart
 - buildings 8
 - grids 8
 - rail 8
 - sewers 8
- Smarter Planet 10, 13, 18
- Smarter Planet Era 12, 16, 25
- Solid® Accelerator API 19
- solidDBEnrich Operator 261
- Solution Framework layer 322
 - Equities Trading 322
 - Options Trading 322
- Sort Operator 204, 224
- Source Editor View 45
- Source Operator 47–48, 188, 204
- Source Stream Operator 57
- Split Operator 99, 204, 216, 219
- SRM 42, 128, 130, 133, 179
- SSH 161
- SSH communication 156, 162
- SSH keys 161
- SSH protocol 159
- standard-based 11
- starting an Instance 134
- Stateful (potentially) Operators 142
- Stateful Operators 142
- Stateless Operators 141
- static data 4, 16
- stopping an Instance 134
- storage performance 2
- stream computing 1, 4, 37
- stream computing platforms 23
- stream joinedRecords 56
- streaming data 1, 4
- Streams 16
 - administrative interfaces 34
 - Application 65
 - Applications 20, 39
 - bundle 203
 - data flow 188
 - development environment 23
 - environment 33
 - installation 304
 - installer program 308
 - modifiers 52
 - performance 146
 - programming interfaces 34
 - runtime environment 48
 - services 127
 - system log 245
 - toolkits 313
 - tools 64
 - topologies 129
 - multiple hosts 132
 - single Host computer 130
 - Tuple 82
 - windows 63, 196
- Streams Adapters Toolkit 48
- Streams Application Graph 70
- Streams Application Graph View 69
- Streams Application Manager (SAM) 42, 128
- Streams Authorization and Authentication Service 42
- Streams Debugger 21, 249
- Streams Functor Operators 120
- Streams Host 41
- Streams Instance 40–41, 72, 128, 246
- Streams Job 246
- Streams Live Graph 21, 152, 182
 - toolkit 19
- Streams Live Graph Outline 153, 182
- Streams Live Graph Outline View 71
- Streams Live Graph View 45, 243
- Streams Mining Toolkit 316
- Streams Policy Macros 44
- Streams Preprocessor 226
- Streams Processing Application Declarative Engine

- (SPADE) 19, 66, 249
 - compiler 175
- Streams Processing Language 19, 39, 57, 189–190, 204
 - Application 69
 - Operators 204
- Streams Processing Language Compiler 58
- Streams Project 66
- Streams Resource Manager (SRM) 42, 128, 130, 133, 145
- Streams Runtime 22, 84, 128, 149
- Streams Scheduler (SCH) 42, 129, 179
- Streams Sink 84
- Streams Studio 19, 21, 241, 248
 - plug-ins 65
- Streams Web Service (SWS) 43, 129, 131, 179
- streamsadmin 133
- streamtool 64, 135, 245
- structured data 2, 82
- surveillance society 13
- System R 18
- System S 18

T

- TCP
 - based data access 81
 - communication 163
- TCP/IP
 - port numbers 162
 - socket 209
- test
 - environment 149
 - environment flexibility 149
 - monitoring tools 150
 - setup cost 149
- testing performance 150
- throttledRate 241
- throughput 144, 152
 - focused applications 78
- Throughput application class 80
- traditional
 - computing 4
 - sources 9
 - stored data 4
- Transaction Processing Performance Council (TPC) 147
- trigger policy 225
- Tumbling windows 61, 197, 200, 216

- Tuples 44
- Typedefs 192

U

- UDOP 86, 110, 118, 121, 142, 145, 176, 276
 - Also see User-defined Operator
- UDP 84
 - based data access 81
- ultra-low latency 24
- uniform resource identifier (URI) 208, 210
- unstructured data 7, 15, 82, 86, 95
 - pattern 109
 - types 109
 - audio streams 109
 - free form text 109
 - video streams 109
- unstructured payload 281
- US Securities and Exchange Commission 20
- User-defined Built-in Operator (UBOP) 111, 118, 122, 142, 176, 289–291
- User-defined function (UDF) 120, 266
- User-defined Operator (UDOP) 86, 276
- User-defined Sink Operator 273
- User-defined Source Operator 268
- using existing analytics 120

V

- ValueCharacterisation stream 94
- Vetted Domains 44
- video streams 98
- Virtual Schema 63
- Virtual Schema Definition 56

W

- Wave Generators 61
- WFOSource Operator 262
- window eviction policy 198
- window trigger policy 198
- windows
 - Sliding 198
 - Tumbling 197
- windows of data 61
- workspace 65
- worldwide internet 11

X

- x-ray films 2



IBM InfoSphere Streams: Harnessing Data in Motion

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



IBM InfoSphere Streams

Harnessing Data in Motion



**Performing complex
real-time analytics
on data in motion**

**Supporting
scalability and
dynamic adaptability**

**Enabling continuous
analysis of data**

In this IBM Redbooks publication, we discuss and describe the positioning, functions, capabilities, and advanced programming techniques for IBM InfoSphere Streams.

Stream computing is a new paradigm. In traditional processing, queries are typically run against relatively static sources of data to provide a query result set for analysis. With stream computing, a process that can be thought of as a continuous query, that is, the results are continuously updated as the data sources are refreshed. So, traditional queries seek and access static data, but with stream computing, a continuous stream of data flows to the application and is continuously evaluated by static queries. However, with IBM InfoSphere Streams, those queries can be modified over time as requirements change.

IBM InfoSphere Streams takes a fundamentally different approach to continuous processing and differentiates itself with its distributed runtime platform, programming model, and tools for developing continuous processing applications. The data streams consumable by IBM InfoSphere Streams can originate from sensors, cameras, news feeds, stock tickers, and a variety of other sources, including traditional databases. It provides an execution platform and services for applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks